# Project Advance Lane Finding

**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 (https://review.udacity.com/#!/rubrics/571/view) 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 (https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/writeup_template.md) is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

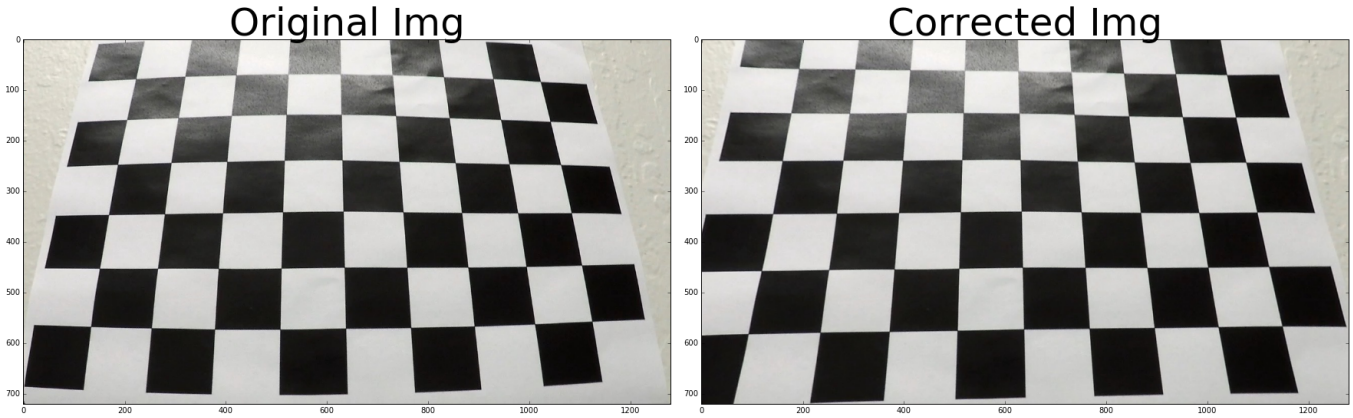
## Camera Calibration

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

This is the first step of the image processing pipline. I started by preparing the `objpt` and `imgpt`. The `objpt` are just points correspond to real-world object, in this case, the corners of each chessboard. The `imgpt` are the (x,y) coordinates of the pixel positions of each chessboard image. The `objpt` are all the same, and will be appended to an array called `objpoints` once it detects a 9 by 6 chessboard. The `imgpt` will also be appended to an array called `imgpoints` once it detects the chessboard.

This piece of code is located at `./myfiles/undistort_img.py`. I made it a class so it can be reused later. I also saved the `imgpoints` and `objpoints` with `pickle` module so they can be reused later.

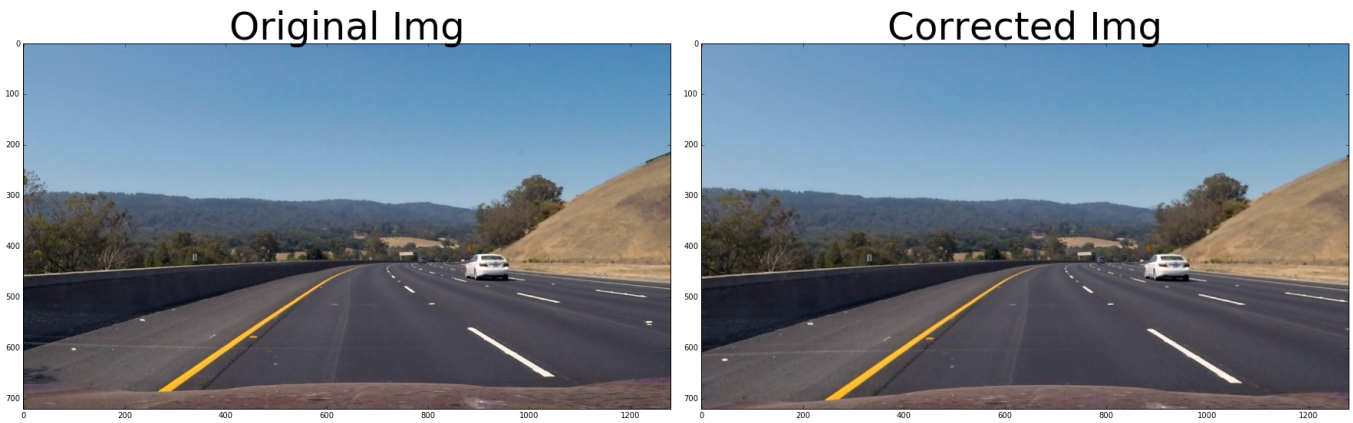
I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:




## Pipeline (single images)

### 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 color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

I applied both color and gradient thresholds to create my thresholded binary image. For the color threshold, I first changed my image's color space from RGB to HSV. Then I setup a yellow colored mask and white colored mask for extract only the the yellow and white color out. I did this because the lanes are either yellow or white.

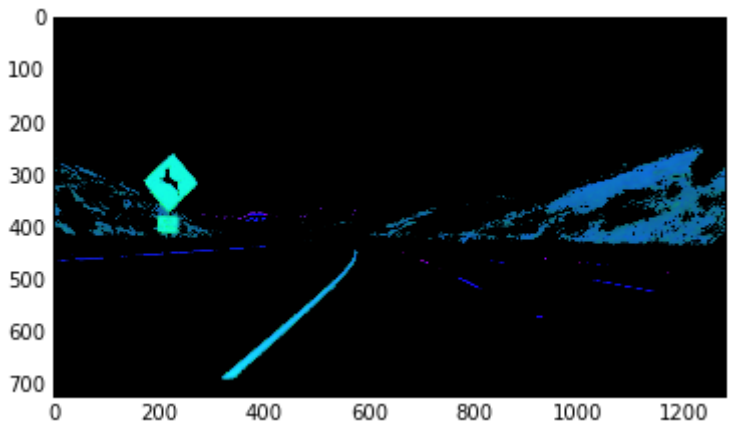
```
def color_thresh(img):
    img = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)

    yellow_min = np.array([15, 100, 120], np.uint8)
    yellow_max = np.array([80, 255, 255], np.uint8)
    yellow_mask = cv2.inRange(img, yellow_min, yellow_max)

    white_min = np.array([0, 0, 200], np.uint8)
    white_max = np.array([255, 30, 255], np.uint8)
    white_mask = cv2.inRange(img, white_min, white_max)

    binary_output = np.zeros_like(img[:, :, 0])
    binary_output[((yellow_mask != 0) | (white_mask != 0))] = 1

    filtered = img
    filtered[((yellow_mask == 0) & (white_mask == 0))] = 0
    #plt.imshow(filtered)

    return binary_output
```

For the gradient thresh part, I compared the gradients of the image in terms of x direction and magnitude, plus the arctan value as well. I setup the `sobel_kernel` to be of size 7 instead of the default 3 because I want the lines on my binary thresholded image to look better.

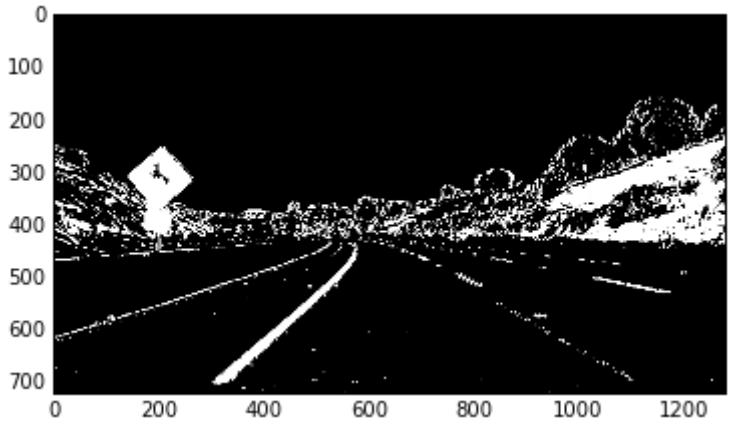
Finally I combined the colored thresholded image with my gradient thresholded image.

```
def sobel_color(self,img):
    sobelimg = self.sobel_thresh(img)
    colorimg = self.color_thresh(img)
    newimg = np.zeros_like(sobelimg)
    newimg[(sobelimg == 1) | (colorimg == 1)] = 1
    return newimg
```

Here's a picture showing how `colorimg`, after masking look like:




Here's a picture showing how `newimg` look like:




This part of code is also implemented as a class. It is located at `./myfiles/threshhold_img.py`,it also depends on these two files `./myfiles/hls.py` and `./myfiles/sobel.py`.

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

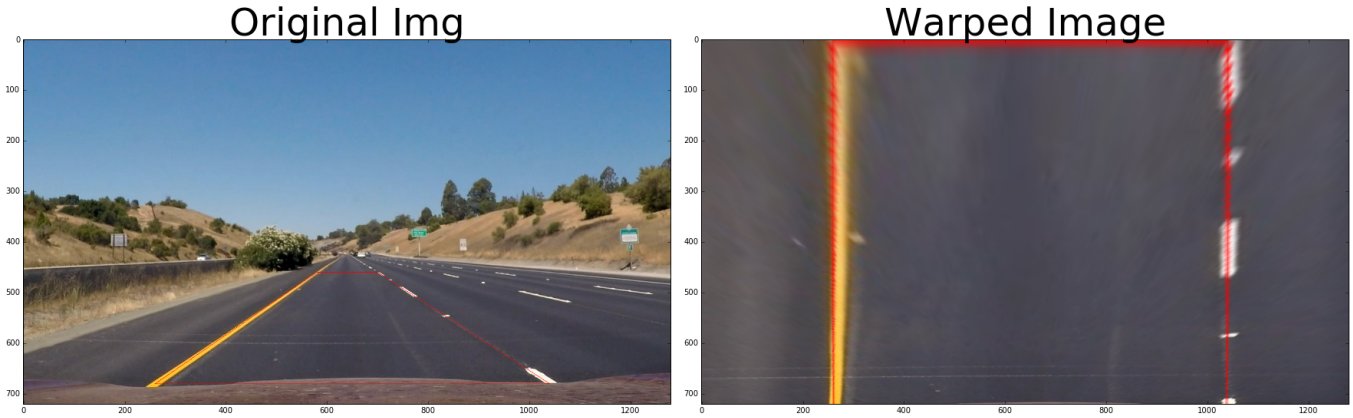
The warping part turns the image to bird-eye view. I predefined the `src` points and `dst` points. The function `cv2.getPerspectiveTransform(self.src, self.dst)` returns `M` which is used to transform the image to bird-eye view. The function `cv2.getPerspectiveTransform(self.dst,self.src)` returns `Minv`, which is used to restore image back to it's original perspect.

```
class warp_img:
    def __init__(self):
        self.src = np.float32([
            [580, 460],
            [700, 460],
            [1040, 680],
            [260, 680],
        ])

        self.dst = np.float32([
            [260, 0],
            [1040, 0],
            [1040, 720],
            [260, 720],
        ])
        self.M = cv2.getPerspectiveTransform(self.src, self.dst)
        self.Minv = cv2.getPerspectiveTransform(self.dst,self.src)
```

This resulted in the following source and destination points:

| Source | Destination |
|---|---|
| 585, 460 | 260, 0 |
| 700, 460 | 1040, 0 |
| 1040, 680 | 1040, 720 |
| 260, 680 | 260, 720 |

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.




This part of code is stored at `./myfiles/warp_img.py`.

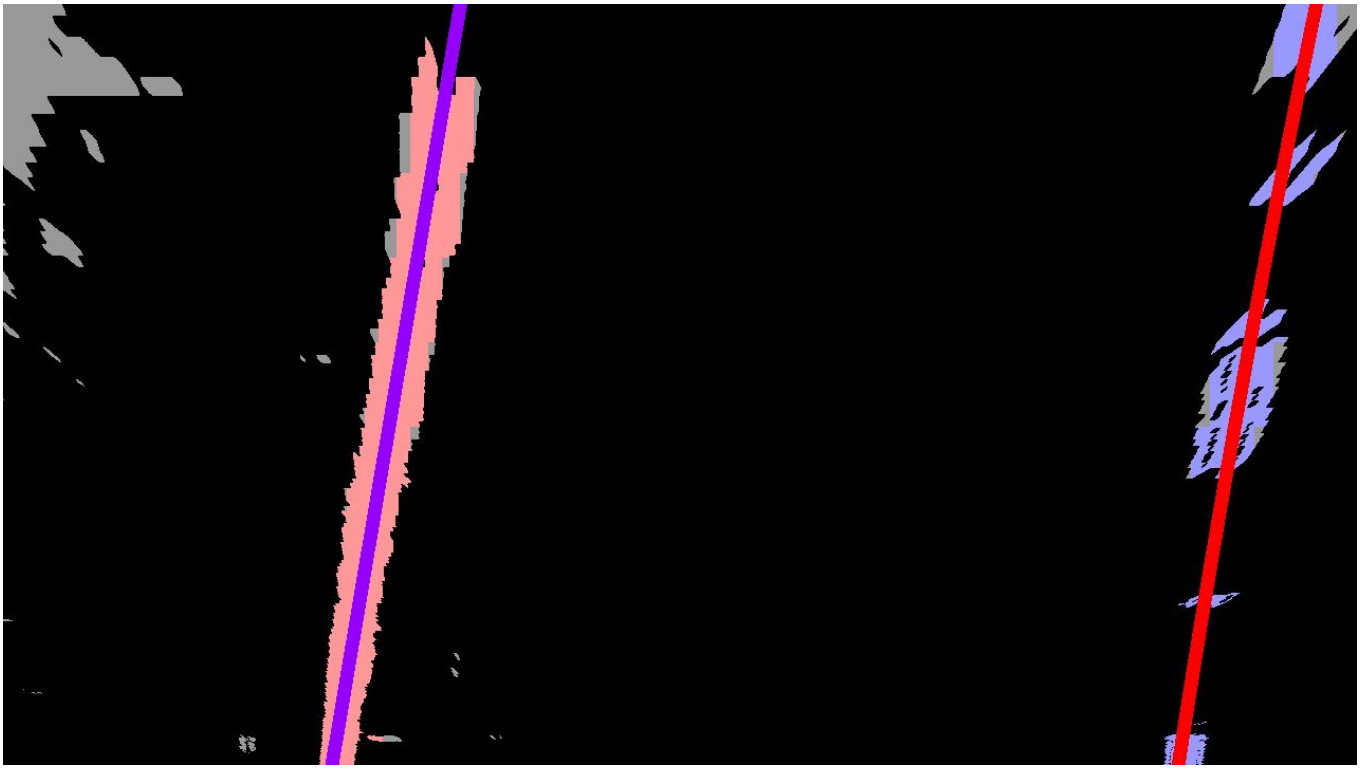
**4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?**

Ì identified the lane-line pixels with a sliding window search that was based on convolution. So the idea is to split the image into horitonal segments and use a window signal with ones to convolute against it. **The peak of the convolved signal is where there was the highest overlap of pixels and the most likely position for the lane marker.** I changed the margin to be 30 which is smaller than the orignal one because I want the lines to be continous so its safter to shrink the margin size.

```
def find_window_centroids(self,warped):

    window_centroids = [] # Store the (left,right) window centroid positions per level
    window = np.ones(self.window_width) # Create our window template that we will use fo
r convolutions
    # First find the two starting positions for the left and right lane by using np.sum
 to get the vertical image slice
    # and then np.convolve the vertical image slice with the window template

    # Sum quarter bottom of image to get slice, could use a different ratio
    l_sum = np.sum(warped[int(3*warped.shape[0]/4):,:int(warped.shape[1]/2)], axis=0)
    l_center = np.argmax(np.convolve(window,l_sum))-self.window_width/2
    r_sum = np.sum(warped[int(3*warped.shape[0]/4):,int(warped.shape[1]/2):], axis=0)
    r_center = np.argmax(np.convolve(window,r_sum))-
self.window_width/2+int(warped.shape[1]/2)

    # Points used to draw all the left and right windows
    l_points = np.zeros_like(warped)
    r_points = np.zeros_like(warped)

    # Add what we found for the first layer
    window_centroids.append((l_center,r_center))
    # apply masking
    l_mask = self.window_mask(warped,window_centroids[0][0],0)
    r_mask = self.window_mask(warped,window_centroids[0][1],0)
    l_points[(l_points == 255) | ((l_mask == 1) ) ] = 255
    r_points[(r_points == 255) | ((r_mask == 1) ) ] = 255


    # Go through each layer looking for max pixel locations
    for level in range(1,(int)(warped.shape[0]/self.window_height)):
        # convolve the window into the vertical slice of the image
        image_layer = np.sum(warped[int(warped.shape[0]-(level+1)*self.window_height):in
t(warped.shape[0]-level*self.window_height),:], axis=0)
        conv_signal = np.convolve(window, image_layer)
        # Find the best left centroid by using past left center as a reference
        # Use window_width/2 as offset because convolution signal reference is at right
 side of window, not center of window
        offset = self.window_width/2
        l_min_index = int(max(l_center+offset-self.margin,0))
        l_max_index = int(min(l_center+offset+self.margin,warped.shape[1]))
        l_center = np.argmax(conv_signal[l_min_index:l_max_index])+l_min_index-offset
        # Find the best right centroid by using past right center as a reference
        r_min_index = int(max(r_center+offset-self.margin,0))
        r_max_index = int(min(r_center+offset+self.margin,warped.shape[1]))
        r_center = np.argmax(conv_signal[r_min_index:r_max_index])+r_min_index-offset
        # Add what we found for that layer
        window_centroids.append((l_center,r_center))
        # Find left and right pts
        l_mask = self.window_mask(warped,window_centroids[level][0],level)
        r_mask = self.window_mask(warped,window_centroids[level][1],level)
        # Add graphic points from window mask here to total pixels found
```

```
        l_points[(l_points == 255) | ((l_mask == 1) ) ] = 255
        r_points[(r_points == 255) | ((r_mask == 1) ) ] = 255


    #return window_centroids,window_centroids_y
    self.window_centroids = window_centroids

    templateL = np.array(l_points,np.uint8)
    templateR = np.array(r_points,np.uint8)

    self.leftpoints = templateL
    self.rightpoints = templateR

    Lpoints = templateL.nonzero()
    Rpoints = templateR.nonzero()

    # find the leftx and lefty
    # templateL = self.leftpoints
    leftx = np.array(Lpoints[1])
    lefty = np.array(Lpoints[0])
    # find the rightx and righty
    # templateR = self.rightpoints
    rightx = np.array(Rpoints[1])
    righty = np.array(Rpoints[0])
    # smooth fit the curve
    self.left_fit = np.polyfit(lefty, leftx, 2)
    self.right_fit = np.polyfit(righty, rightx, 2)
    ploty = self.smoothy
    self.left_fitx = self.left_fit[0]*ploty**2 + self.left_fit[1]*ploty +
self.left_fit[2]
    self.right_fitx = self.right_fit[0]*ploty**2 + self.right_fit[1]*ploty + self.right_
fit[2]
```

As you can see the code also smooth out the line for both the left lane and right line. The code calls the `window_mask` to search for points in the original image that corresponds to the little boxes of window it found. Each `window_centroid` corresponds to a window of size 80x60. Here's how it look like:



You can see that the purple line fits the left lane and red line fits the right lane.

The code is also implemented as class. Although I think I could split up this file into two classes. Too many functions are in this class. This part of code is stored at `./myfiles/line_test1.py`.

**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.**

I calcute the radius of the curvature and the offset by following through Udacity's guide. I noticed that there's a considerable difference between my left and right curvature so I computed the weighted average of them.

```
def getcurvature(self):
    ploty = self.smoothy
    left_fit_cr = np.polyfit(ploty*self.ym_per_pix, self.left_fitx*self.xm_per_pix, 2)
    right_fit_cr = np.polyfit(ploty*self.ym_per_pix, self.right_fitx*self.xm_per_pix, 2)
    # bottom of image
    # y_eval = np.max(ploty)
    y_eval = ploty[6*len(ploty)/7-1:len(ploty)]
    # calculate new curvature
    self.left_curverad = (((1 + (2*left_fit_cr[0]*y_eval*self.ym_per_pix +
left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])).mean()
    self.right_curverad = (((1 + (2*right_fit_cr[0]*y_eval*self.ym_per_pix + right_fit_c
r[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])).mean()
    self.curvature = max(self.left_curverad,self.right_curverad)*0.75+min(self.left_curv
erad,self.right_curverad)*0.25
```

Also the offset:

```
def caloffset(self):
    lane_center = (self.left_fitx[-1]+self.right_fitx[-1])/2
    img_center = 640
    self.offset = (lane_center-img_center)*self.xm_per_pix
```

Since everything is in pixel level I have to convert the distances back into meters level. I set `xm_per_pix = 3.7/700` and `ym_per_pix = 30/720`. This part of code is stored at `./myfiles/line_test1.py`.

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

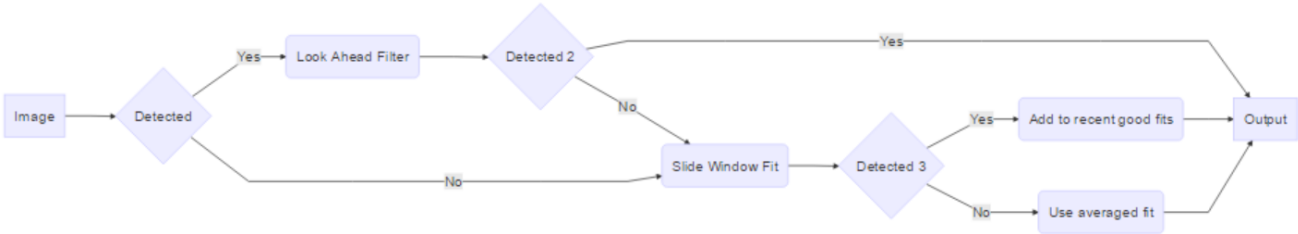
I implemented this step in lines 69 through 92 in my code in `./myfiles/line_test1.py` in the class `line`. Here is an example of my result on a test image:

---

## 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 graph of my video pipeline:




I would think that the "Look Ahead Filter" would accelerate the progress of rendering a video. But I haven't tested how much time would be prolonged if taken away.

I included the video file `./myfiles/project_video_rendered.mp4` in the zip file.

---

## 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?**

For this project, I think multiple problems were faced. One of them is finding the correct color space to work with. I ended up masking the white and yellow color instead of only using S channel to mask it. Another difficult part I think was how to decide if an image was detected. I mean I compared the slopes of two lines, the curvature of two lines and the horizontal distance between two lines.

My pipeline is prone to fail at places that have shadows. Perhaps implementing a shadow removing algorithm will improve the overall performance. I also think haze removal is needed as well, I heard it is somehow related to the "dark channel" in an image?