# 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.

# Writeup / README

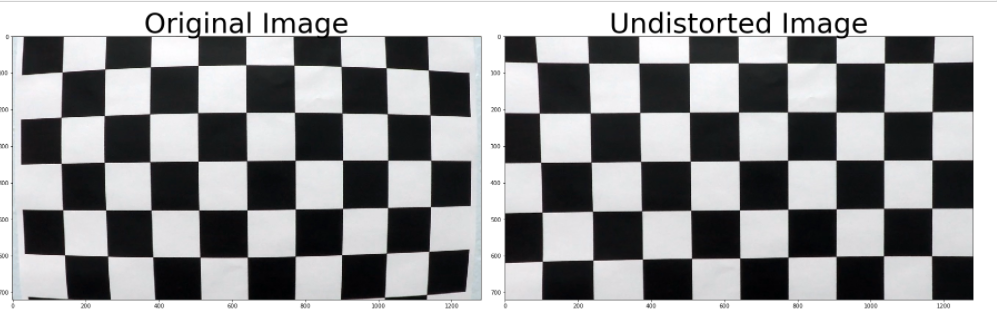
## Camera Calibration

### 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 “Step1: Camera Calibration” of the IPython notebook located in "CarND-Advanced-Lane-Lines-Rober\_v01.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:



## Pipeline (single images)

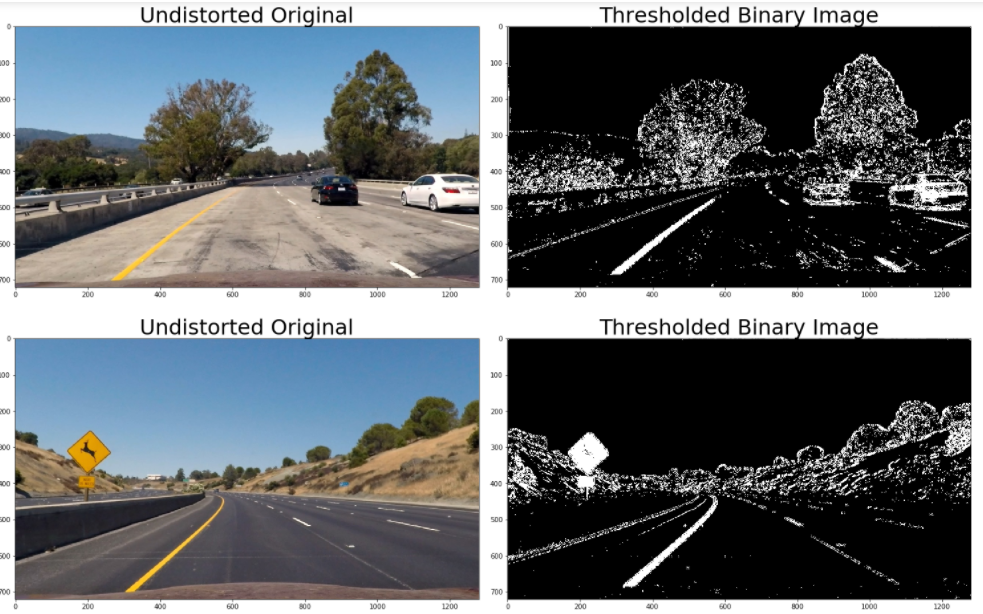
### 1. Provide an example of a distortion-corrected image.

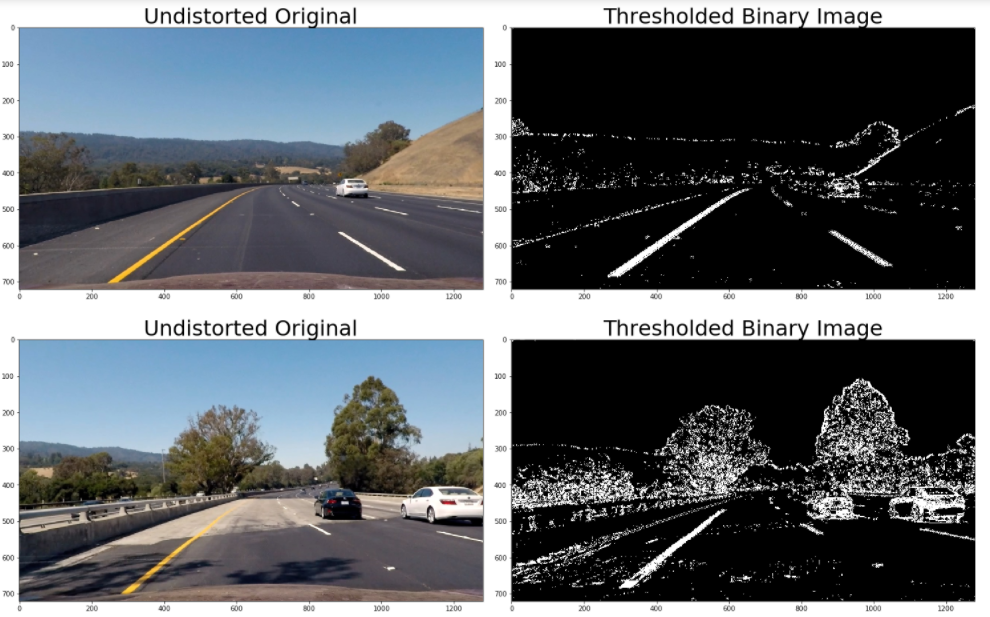
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:



### 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 used a combination of color and gradient thresholds to generate a binary image (thresholding steps & functions in file `thresholding.py`). Here's an example of my output for this step.





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

The code for my perspective transform includes a function called `warper()`, which appears in the file `lines.py` .The `warper()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

```python

src = np.float32(

[[(img\_size[0] / 2) - 55, img\_size[1] / 2 + 100],

[((img\_size[0] / 6) - 10), img\_size[1]],

[(img\_size[0] \* 5 / 6) + 60, img\_size[1]],

[(img\_size[0] / 2 + 55), img\_size[1] / 2 + 100]])

dst = np.float32(

[[(img\_size[0] / 4), 0],

[(img\_size[0] / 4), img\_size[1]],

[(img\_size[0] \* 3 / 4), img\_size[1]],

[(img\_size[0] \* 3 / 4), 0]])

```

This resulted in the following source and destination points:

| Source | Destination |

|:-------------:|:-------------:|

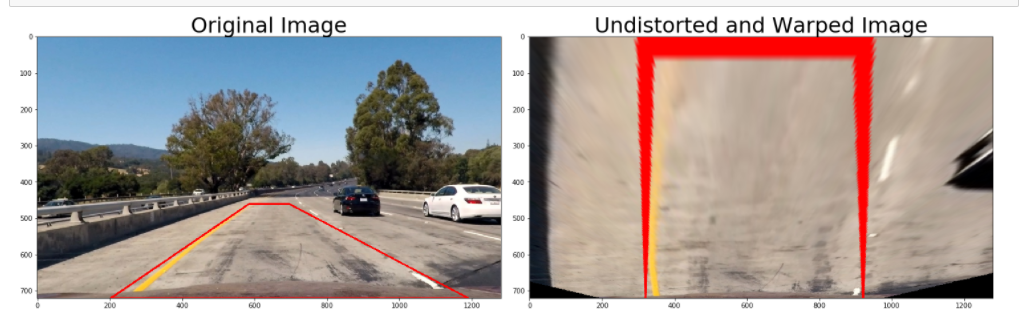
| 585, 460 | 320, 0 |

| 203, 720 | 320, 720 |

| 1127, 720 | 960, 720 |

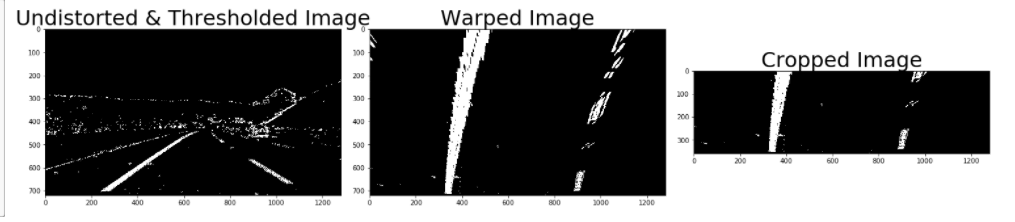
| 695, 460 | 960, 0 |

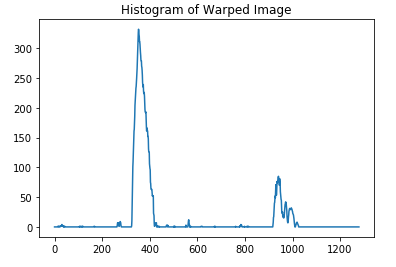
I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



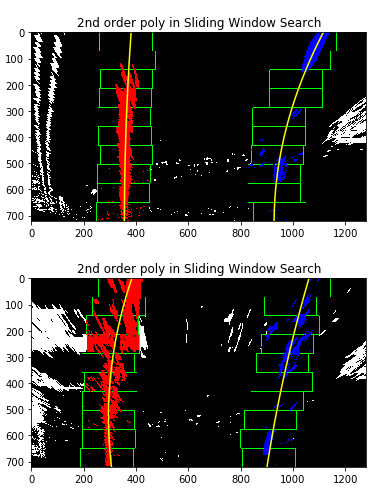
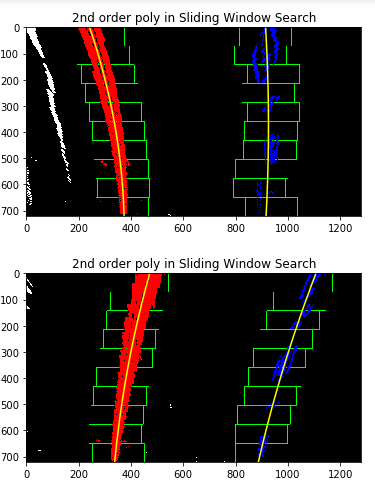
### 4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

I first take a **histogram** along all the columns in the *lower half* of the image like this:





Then I use ‘fit\_poly()’ function, ()`, which appears in the file `lines.py`, to fit my lane lines with a 2nd order polynomial kinda like this.



### 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 follow the code from Udacity tutorial for this task.

‘’’

def find\_curvature(ploty, leftx, rightx):

# Define conversions in x and y from pixels space to meters

ym\_per\_pix = 30/720 # meters per pixel in y dimension

xm\_per\_pix = 3.7/700 # meters per pixel in x dimension

# Fit new polynomials to x,y in world space

left\_fit\_cr = np.polyfit(ploty\*ym\_per\_pix, leftx\*xm\_per\_pix, 2)

right\_fit\_cr = np.polyfit(ploty\*ym\_per\_pix, rightx\*xm\_per\_pix, 2)

# Calculate the new radii of curvature

left\_curverad = ((1 + (2\*left\_fit\_cr[0]\*y\_eval\*ym\_per\_pix + left\_fit\_cr[1])\*\*2)\*\*1.5) / np.absolute(2\*left\_fit\_cr[0])

right\_curverad = ((1 + (2\*right\_fit\_cr[0]\*y\_eval\*ym\_per\_pix + right\_fit\_cr[1])\*\*2)\*\*1.5) / np.absolute(2\*right\_fit\_cr[0])

# Now our radius of curvature is in meters

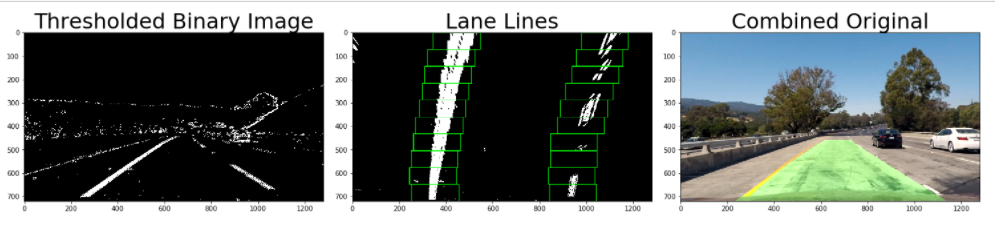
print(left\_curverad, 'm', right\_curverad, 'm')

return (left\_curverad,right\_curverad)

’’’

### 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 # through # in my code in `yet\_another\_file.py` in the function `map\_lane()`. 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 [link to my video result](./project\_video.mp4)

---

## 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?

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

# RUBRIC POINTS

[//]: # (Image References)

[image1]: ./examples/undistort\_output.png "Undistorted"

[image2]: ./test\_images/test1.jpg "Road Transformed"

[image3]: ./examples/binary\_combo\_example.jpg "Binary Example"

[image4]: ./examples/warped\_straight\_lines.jpg "Warp Example"

[image5]: ./examples/color\_fit\_lines.jpg "Fit Visual"

[image6]: ./examples/example\_output.jpg "Output"

[video1]: ./project\_video.mp4 "Video"

## [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.

---

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.image as mpimg

import glob

import cv2

# Read in a thresholded image

warped = mpimg.imread('warped\_example.jpg')

# window settings

window\_width = 50

window\_height = 80 # Break image into 9 vertical layers since image height is 720

margin = 100 # How much to slide left and right for searching

def window\_mask(width, height, img\_ref, center,level):

output = np.zeros\_like(img\_ref)

output[int(img\_ref.shape[0]-(level+1)\*height):int(img\_ref.shape[0]-level\*height),max(0,int(center-width/2)):min(int(center+width/2),img\_ref.shape[1])] = 1

return output

def find\_window\_centroids(image, window\_width, window\_height, margin):

window\_centroids = [] # Store the (left,right) window centroid positions per level

window = np.ones(window\_width) # Create our window template that we will use for 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))-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))-window\_width/2+int(warped.shape[1]/2)

# Add what we found for the first layer

window\_centroids.append((l\_center,r\_center))

# Go through each layer looking for max pixel locations

for level in range(1,(int)(warped.shape[0]/window\_height)):

# convolve the window into the vertical slice of the image

image\_layer = np.sum(warped[int(warped.shape[0]-(level+1)\*window\_height):int(warped.shape[0]-level\*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 = window\_width/2

l\_min\_index = int(max(l\_center+offset-margin,0))

l\_max\_index = int(min(l\_center+offset+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-margin,0))

r\_max\_index = int(min(r\_center+offset+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))

return window\_centroids

window\_centroids = find\_window\_centroids(warped, window\_width, window\_height, margin)

# If we found any window centers

if len(window\_centroids) > 0:

# Points used to draw all the left and right windows

l\_points = np.zeros\_like(warped)

r\_points = np.zeros\_like(warped)

# Go through each level and draw the windows

for level in range(0,len(window\_centroids)):

# Window\_mask is a function to draw window areas

l\_mask = window\_mask(window\_width,window\_height,warped,window\_centroids[level][0],level)

r\_mask = window\_mask(window\_width,window\_height,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

# Draw the results

template = np.array(r\_points+l\_points,np.uint8) # add both left and right window pixels together

zero\_channel = np.zeros\_like(template) # create a zero color channel

template = np.array(cv2.merge((zero\_channel,template,zero\_channel)),np.uint8) # make window pixels green

warpage = np.array(cv2.merge((warped,warped,warped)),np.uint8) # making the original road pixels 3 color channels

output = cv2.addWeighted(warpage, 1, template, 0.5, 0.0) # overlay the orignal road image with window results

# If no window centers found, just display orginal road image

else:

output = np.array(cv2.merge((warped,warped,warped)),np.uint8)

# Display the final results

plt.imshow(output)

plt.title('window fitting results')

plt.show()