SDCND - P4 : Advanced Lane Detection

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.

Camera Calibration

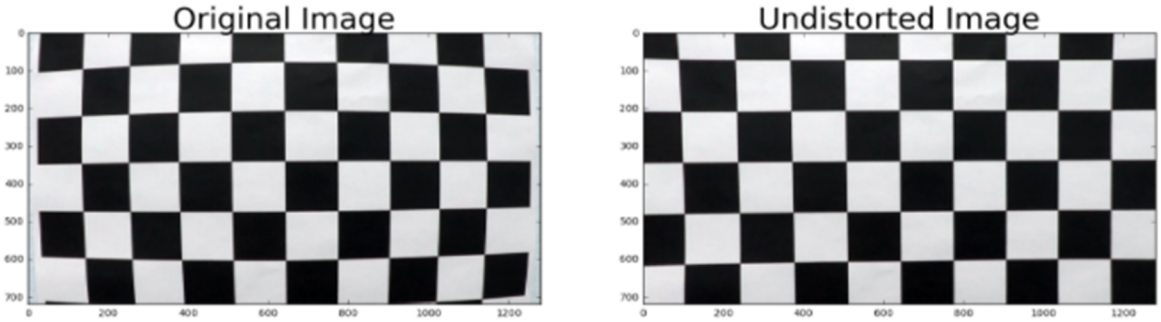
1. Have the camera matrix and distortion coefficients been computed correctly and checked on one of the calibration images as a test?

In order to do camera calibration, I defined a .py file named as 'camerCalibration.py' where I defined the a function '*calibrate_camera(file_path)*' for calibrating the camera. The images given in the camera_cal directory were used for calibrating the camera. Here is my function:

```
# Define camera calibration
def calibrate_camera(file_path):
    objpoints = []            ##3D points in real world space
    imgpoints = []            ##Correspoind 2D points in image plane
    # Prepare objects points
    objp = np.zeros((6*9,3), np.float32)
    objp[:,:2] = np.mgrid[0:9, 0:6].T.reshape(-1,2)
    # Get the image/s
    images = glob.glob(file_path)
    # Iterate over the files and apply processing steps
    for image in images:
        img = cv2.imread(image)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # Find chessboard corners
        ret, corners = cv2.findChessboardCorners(gray,(9,6), None)
        # If the corners are found, add the values to imgpoints and objpoints
        if ret == True:
            objpoints.append(objp)
            imgpoints.append(corners)

            # Draw and (optional) dispaly the corners
            img = cv2.drawChessboardCorners(img, (9,6), corners, ret)
            #plt.imshow(img)
            #plt.show()
        else:
            continue
    # Calibrate the camera
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)
    return ret, mtx, dist, rvecs, tvecs
```

Fig. Code lines (12-42) in cameraCalibration.py

All the images were converted to grayscale before finding the chess board corners. For the images, where the chess board corners were located successfully, corresponding image points and object points were stored in the list which were then used to calibrate the camera by passing them into the '*cv2.calibrateCamera()*' function.



Pipeline (single images)

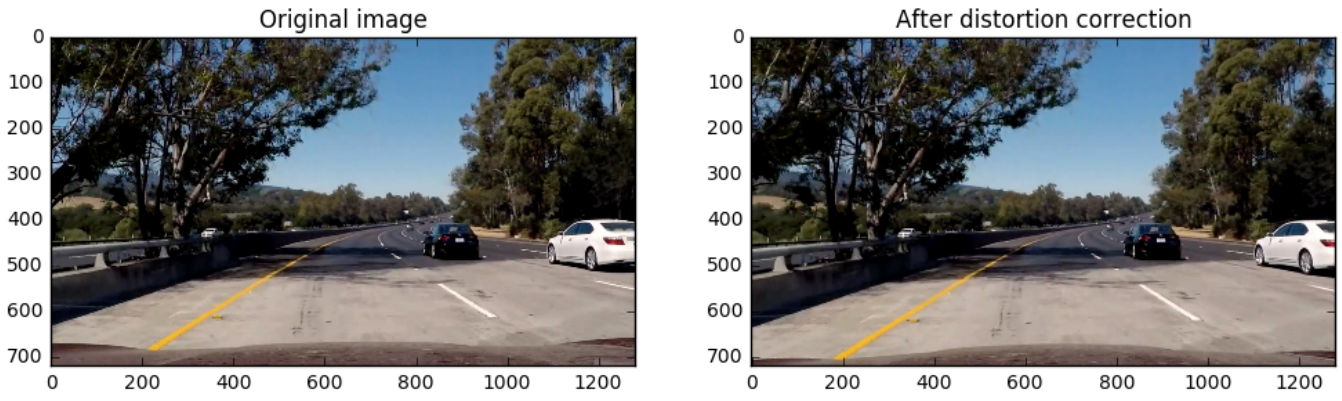
1. Has the distortion correction been correctly applied to each image?

In order to correct the distortion, I defined a function *undistort_image()* inside the *cameraCalibration.py* file. The function is show below:

```
# Define undistortion procedure
def undistort_image(img, matrix, dist_coeff):
    undistorted_image = cv2.undistort(img, matrix, dist_coeff, None, matrix)
    #plt.imshow(undistorted_image)
    return undistorted_image
```

Fig. Code lines (48-51) in cameraCalibration.py

The function takes an image along with calibration matrix and distortion coefficients as the input parameters and returns an undistorted image. For example, when tested on the *test5.jpg*, the result looks like this:



2. Has a binary image been created using color transforms, gradients or other methods?

In order to binary threshold, I tried different techniques like I applied only sobel operator and no color threshold, only color threshold and no sobel, both color and gradient threshold. After checking the results, I found out that combined threshold gives the best results. Also, I found that applying sobel on V channel in HSV color space gives much better results as compared to S channel in the HLS color space. For color threshold, I used a combination of V channel and R channel. The functions for finding the gradient threshold, color threshold and combined threshold are defined in *cameraCalibration.py* file as shown below:

```
# Absolute Sobel threshold
def abs_sobel_thresh(img, orient='x', sobel_kernel =5 , thresh=(25,125)):
    # Convert the image to HSV
    image = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    gray = image[:,:,2]

    # Apply sobel
    if orient == 'x':
        gradient = cv2.Sobel(gray, cv2.CV_64F, 1,0, sobel_kernel)
    else:
        gradient = cv2.Sobel(gray, cv2.CV_64F, 0,1, sobel_kernel)

    # Take absolute
    abs_gradient = np.absolute(gradient)
    # Scale the image
    scaled_img = np.uint8(255*abs_gradient/np.max(abs_gradient))
    # Define a mask
    binary_output = np.zeros_like(scaled_img)
    binary_output[(scaled_img >= thresh[0]) & (scaled_img <= thresh[1])] =1
    return binary_output
```

Fig. Code lines (79-97) in cameraCalibration.py

```python
# Color Thresholding
def color_threshold(img, color_thresh=(200,255)):
    # Convert to HSL space
    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    # Separate the s channel
    v_channel = hsv[:,:,2]
    r_channel = img[:,:,0]
    # Create and return a binary image
    v_binary = np.zeros_like(v_channel)
    r_binary = np.zeros_like(r_channel)
    v_binary[(v_channel >= color_thresh[0]) & (v_channel <= color_thresh[1])] = 1
    r_binary[(r_channel >= color_thresh[0]) & (r_channel <= color_thresh[1])] = 1
    # Combine thresholding of v and r channels
    comb_binary =  cv2.bitwise_or(v_binary, r_binary)
    return comb_binary

# Define combined thresholding for color, gradient,etc...
def combined_binary_threshold(img):
    # Find gradient thresholds
    gradx = abs_sobel_thresh(img, orient='x', sobel_kernel=7, thresh=(50,150))
    # Combine all the gradient thresholds
    gmd_img = np.zeros_like(gradx)
    gmd_img[(gradx == 1)] =1
    # Find color threshold
    color_thresh_img = color_threshold(img)
    # Stack the color and gradient thresholds
    color_binary = np.dstack((np.zeros_like(gmd_img), gmd_img, color_thresh_img))
    # Combine the color and gradient thresholds
    combined_binary = np.zeros_like(gmd_img)
    combined_binary[(gmd_img ==1) | (color_thresh_img == 1)] = 1
    return combined_binary,  color_binary
```

Fig. Code lines (103-145) in cameraCalibration.py

The output of the binary threshold are shown below:

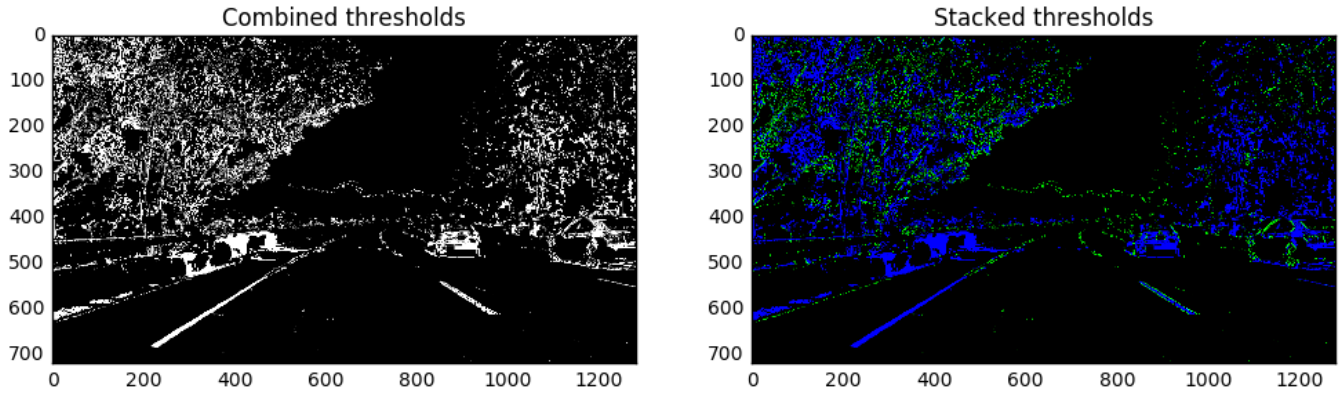


Fig. Binary threshold

3. Has a perspective transform been applied to rectify the image?

In order to catch a bird's eye view, perspective transformation was applied. A function *perspective_transform(img)* is defined in *cameraCalibration.py* file and it looks like this :

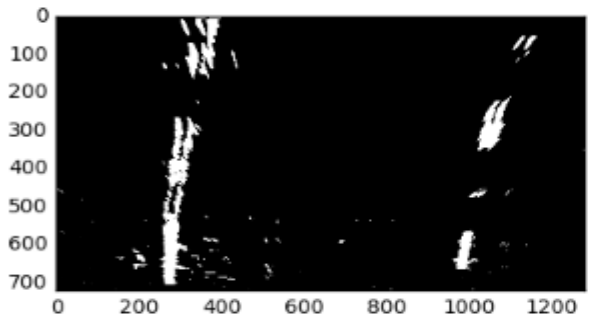
```
# Define perspective transform
def perspective_transform(img):
    #gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # Define source and destination co-ordinates
    src_cord = np.float32(([[240,719],[579,450],[712,450],[1165,719]]))
    dst_cord = np.float32([[300,719], [300,0], [900,0], [900,719]])

    # Find the perspective transformation matrix and inverse transformation matrix
    M = cv2.getPerspectiveTransform(src_cord, dst_cord)
    Minv = cv2.getPerspectiveTransform(dst_cord, src_cord)

    img_size = (img.shape[1], img.shape[0])

    # Warp the image
    warped_image = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)
    return warped_image, M, Minv
```

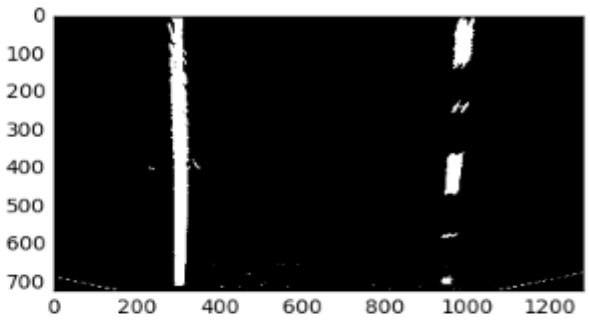
Fig. Code lines (57-72) in cameraCalibration.py

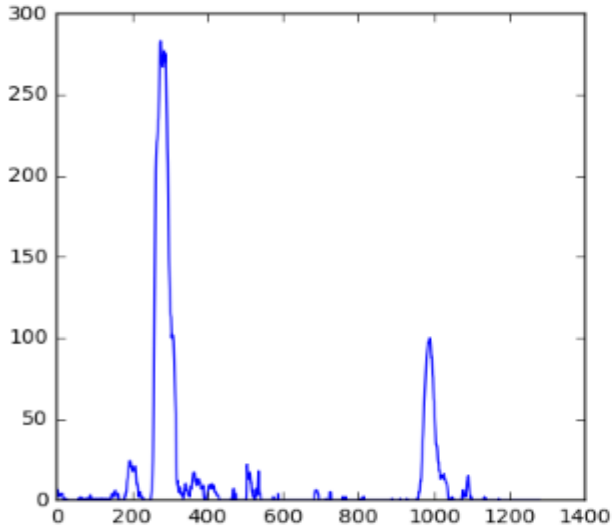
The function takes an image as input parameter. The source co-ordinates and destination-coordinates were hardcoded and found suitable for the whole pipeline. The function performs a perspective transformation on the image and returns three values, the warped image, perspective transformation matrix (M) and inverse perspective transformation matrix (Minv). The result of the perspective transformation are shown below:

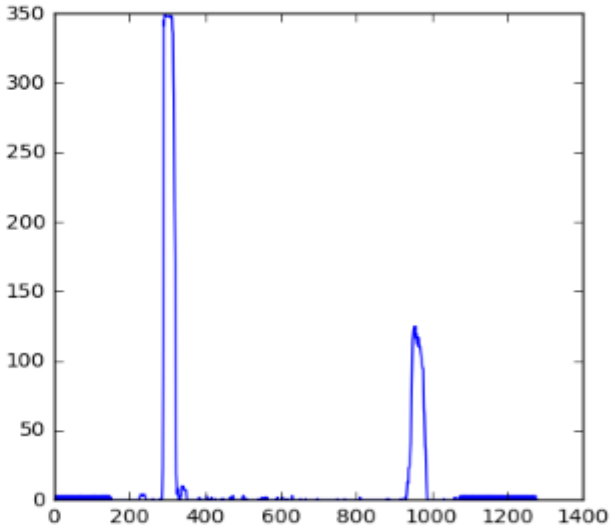
4. Have lane line pixels been identified in the rectified image and fit with a polynomial?

In order to find the lane line pixels, first a histogram of the binary warped image was drawn in order to check if the peaks for both the lanes is distinguishable from the rest of the points in the image. The lanes were located where there are two peaks (one on the left side and one on the right side). In order to get a very clear picture of the peaks, threshold was taken in a way such that the noise is at minimum level in the binary warped image. The results looks like this:









After it was confirmed that we are getting a clear cut picture of the peaks, a sliding window method was adopted in order to capture the lanes and fit a second order degree polynomial across the lanes. The process was then further improved in terms of smoothing of lane detection using the knowledge of previous frame. This whole procedure is defined in *ad_Lanes.py* file which consists a class Lane having appropriate methods for this procedure. The final results are shown below:

* *The *ad_Lanes.py* is documented at each step for reference.

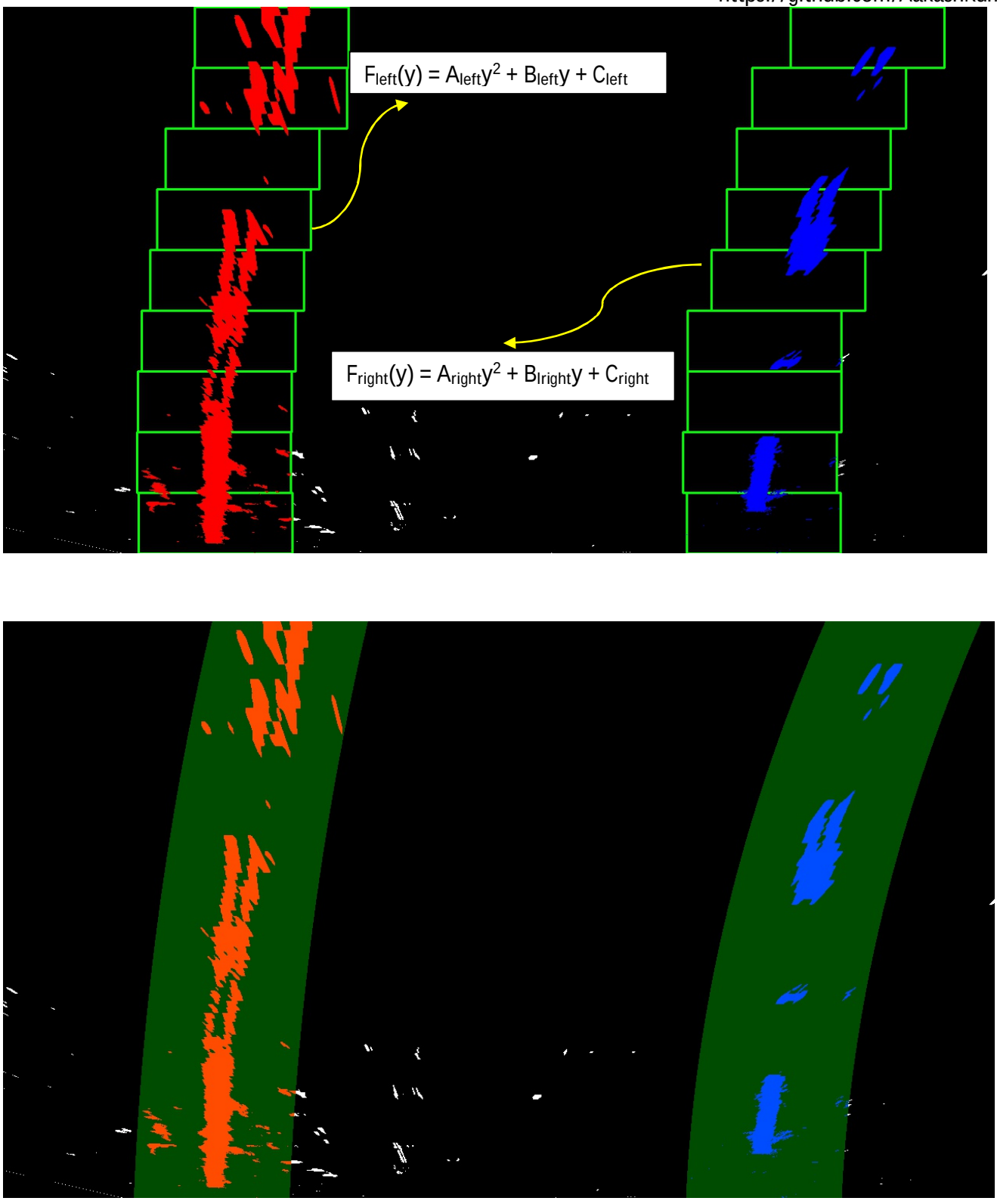


Fig. Lanes detected and fitted with a second order polynomial

5. Having identified the lane lines, has the radius of curvature of the road been estimated? And the position of the vehicle with respect to center in the lane?

Yes, the radius of curvature was calculated accordingly. The radius of curvature is defined as:

$$\text{Radius of curvature} = \frac{\left[1 + \left(\frac{dy}{dx}\right)^2\right]^{3/2}}{\left|\frac{d^2y}{dx^2}\right|}$$

The code for calculating the radius of curvature can be found in *ad_Lanes.py* file in the lines (215-240). The snippet for the same code is shown below:

```
215     def fit_lane(self, img):
216         # Extract the list of x and y coords that are non-zero pixels
217         xycoords = np.nonzero(img)
218         x_arr = xycoords[1]
219         y_arr = xycoords[0]
220 
221         # Fit a second order polynomial to each fake lane line
222         fit = np.polyfit(y_arr, x_arr, deg=2)
223         fitx = fit[0]*y_arr**2 + fit[1]*y_arr + fit[2]
224 
225 
226         fitx = np.insert(fitx, 0, fit[0]*0**2 + fit[1]*0 + fit[2])
227         fity = np.insert(y_arr, 0, 0)
228         fitx = np.append(fitx, fit[0]*719**2 + fit[1]*719 + fit[2])
229         fity = np.append(fity, 719)
230 
231         # Define conversions in x and y from pixels space to meters
232         ym_per_pix = 30/720 # meters per pixel in y dimension
233         xm_per_pix = 3.7/700 # meteres per pixel in x dimension
234         y_eval = np.max(y_arr)
235         fit_cr = np.polyfit(y_arr*ym_per_pix, x_arr*xm_per_pix, 2)
236         fitx_cr = fit_cr[0]*(y_arr*ym_per_pix)**2 + fit_cr[1]*y_arr*ym_per_pix + fit_cr[2]
237 
238         # Get radius of curvature
239         roc = ((1 + (2*fit_cr[0]*y_eval*ym_per_pix + fit_cr[1])**2)**1.5) /np.absolute(2*fit_cr[0])
240         return fit, fitx, fity, roc
```

The position of vehicle with respect to center in the lane was calculated as shown below:

```
338         perfect_center = 1280/2.
339         lane_x = self.last_right_x - self.left_x
340         center_x = (lane_x / 2.0) + self.left_x
341         cms_per_pixel = 370.0 / lane_x   # US regulation lane width = 3.7m
342         dist_from_center = (center_x - perfect_center) * cms_per_pixel
```

The code for above can be found in the *ad_Lanes.py* file in the lines (337-344). The final results look like as shown in the figure below:

Fig. ROC and distance of the vehicle from the center of the lane

Pipeline (video)

1. Does the pipeline established with the test images work to process the video?

Yes, the pipeline work on the video as expected. Here is the link to my video result.

https://youtu.be/4H_3AmFAO-Q

README

1. Has a README file been included that describes in detail the steps taken to construct the pipeline, techniques used, areas where improvements could be made?

Yes, you are reading it!!

Discussion:

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.

I took the same approach as discussed in the lectures except that I made some changes to it. For example, I found that doing a gradient threshold in on V channel in HSV color space provide more robust lane detection results as compared to the S channel in HSL space. Similarly for color threshold, I found out that a combination of color threshold on V channel and R channel works better. Though I feel that both these procedures still need fine tuning as the pipeline fails on the challenge video after a while. My best guess is that my masking is still not working very well in the lightning conditions (too much of light on the road or too much of shadow on the road). In such cases the polyfit is very likely to fail or perform very poorly.

In order to improve the pipeline, so that it works well on challenge as well as harder challenge videos, I need to refine my masking procedure. Though, I have applied conditions in my pipeline when to accept and when to discard a line, but I feel that the pipeline can be further improved if I implement a proper outlier detection procedure. Removing outliers will give a huge improvement to the pipeline.

Finally, given more time, I would have loved to implement a more robust outlier detection method in order to make my algorithm more robust to jitter so that it can work well on the challenge and harder challenge videos too.