Project 4: Advanced Lane Finding

Xavier Sellart Ortega

Contents

1. Objectives of the project

The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- 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.

2. Development process

In this section I will explain how I've achieved the objectives of the project. The process for meeting the objectives can be divided in the following sections:

- Camera Calibration
- Thresholded binary image
- Perspective Transformation
- Lanes detection

2.1. Camera Calibration

The main purpose of this code is to remove the distortion of the camera by using the chessboard images which are located in the "camera_cal" folder of the project.

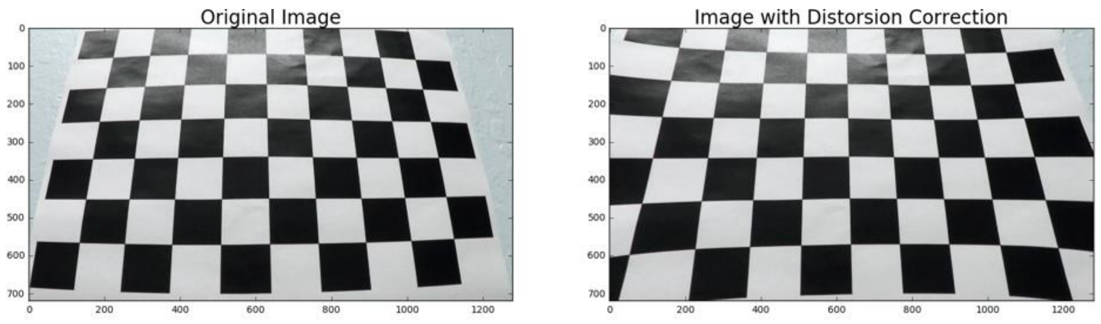
It's important to know that to use the opencv functions for do this calibration it's necessary to know the number of interior corners of the chessboard. In our case the number of horizontal corners is 9 and the number of vertical corners 6.

Firstly, I've converted the image chessboard to grayscale and I looked for the interior corners with the OpenCV function:

cv2 .findChessboardCorners (gray,(nx,ny),None), with this function I was able to find the 2D points of the corners in the images.

So, with the points in the 2D test images and the ones in the 3D real world, I used the following OpenCV function:

cv2.calibrateCamera(objpoints,imgpoints, img_size,NoneNone), which was returning the calibration matrix and distortion parameters required to undistort the image. Once I had this parameters, I used the undistort function from OpenCV, obtaining the following results:



2.2. Thresholded binary image

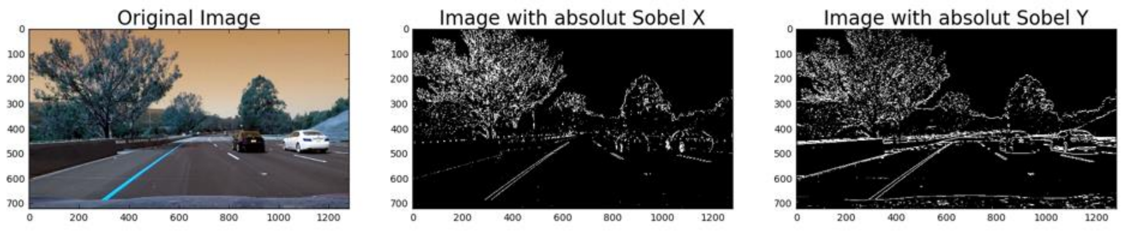
The goal of this part of the development process was to be able to only get the necessary information from the road for detecting the lanes. So, it was necessary to apply several image processing techniques.

2.2.1. Absolute Sobel (x and y)

For applying sobel thresholding to the image, I created a function which was able to apply such type of filter by defining if it was for x or y coordinates, and also the minimum and maximum values which were going to be applied. Here is the body of the function:

***def** abs_sobel_thresh(img, orient='x', thresh_min=0, thresh_max=255):*

And this is the result of applying this threshold:

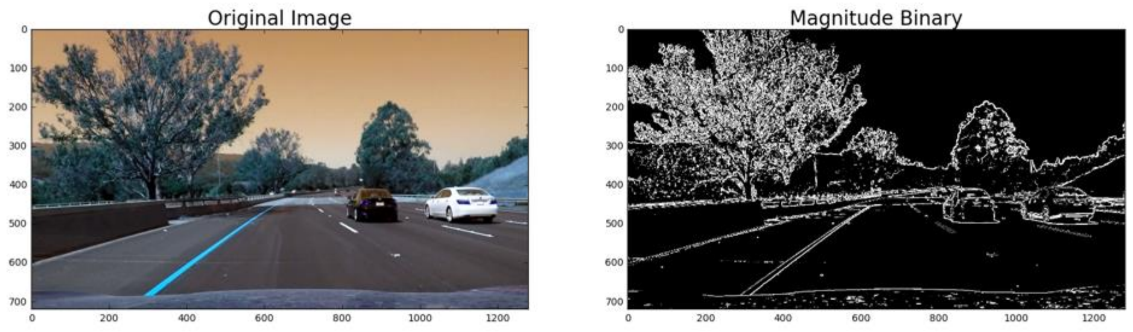


2.2.2. Magnitude Thresholding

A function was generated for applying magnitude thresholding:

def mag_thresh(img, sobel_kernel=3, mag_thresh=(0, 255)):

And the following result was obtained:

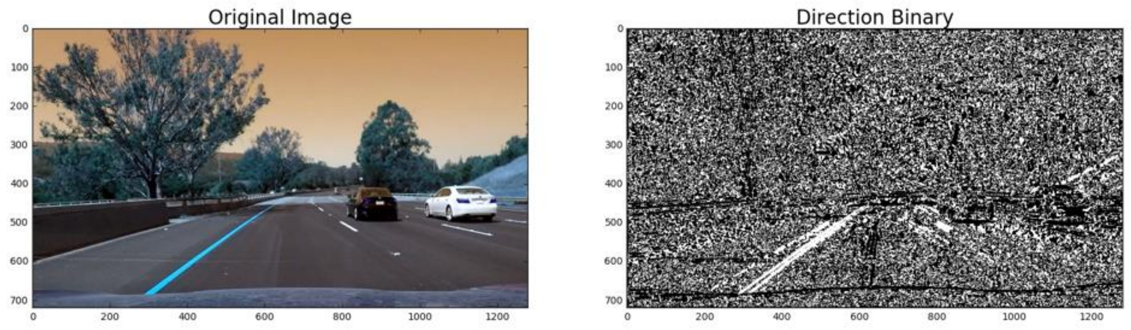


2.2.3. Direction Thresholding

A function was generated for applying direction thresholding:

```
def dir_threshold(img, sobel_kernel=3, thresh=(0, np.pi/2)):
```

And the following result was obtained:



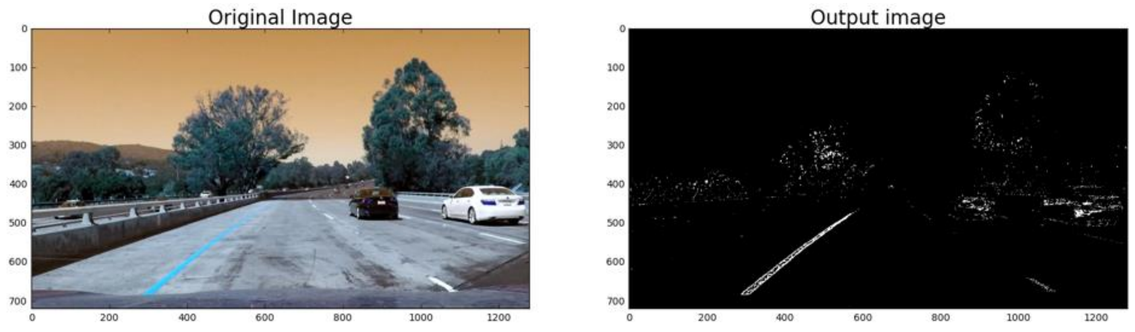
2.2.4. Colour Thresholding

A function was generated for applying colour thresholding:

```
def pipeline_color(img)
```

In that function, I'm transforming the RGB image to HSV and using the Saturation space for thresholding the image.

And the following result was obtained:



2.2.5. Joining Thresholds

For joining the thresholds which have been generated in the project, I firstly made one pipeline for the gray image and other for the colour one.

The logic for the gray pipeline image was:

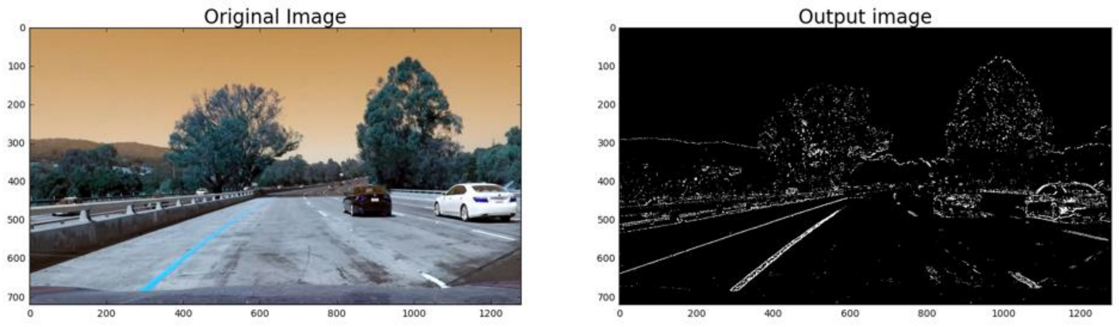
- Firstly, I've transformed the colour image in gray.
- Next step, I've applied the generated functions for gray (absolute sobel (x and y) / magnitude threshold / direction gradient) in the gray image.
- After this step, I was having 4 arrays (one for each threshold), and applied the following logic for each value of the array:

thresholds_all[((abs_gradx == 1) & (abs_grady == 1)) | ((mag == 1) & (dir_grad == 1))] = 1

For the colour logic pipeline, I just used the previous function "pipeline_color".

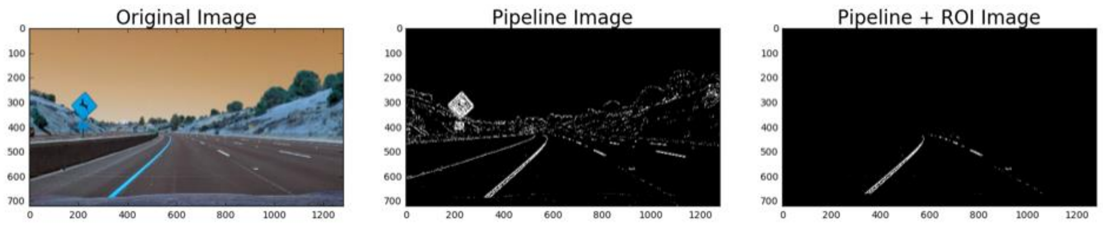
So, after having both pipelines applied (colour and gray), I joined it with a function which was checking if any of the pixels in the filters was 1, the result was also 1.

The following image shows the result:



2.2.6. ROI (Region Of Interest)

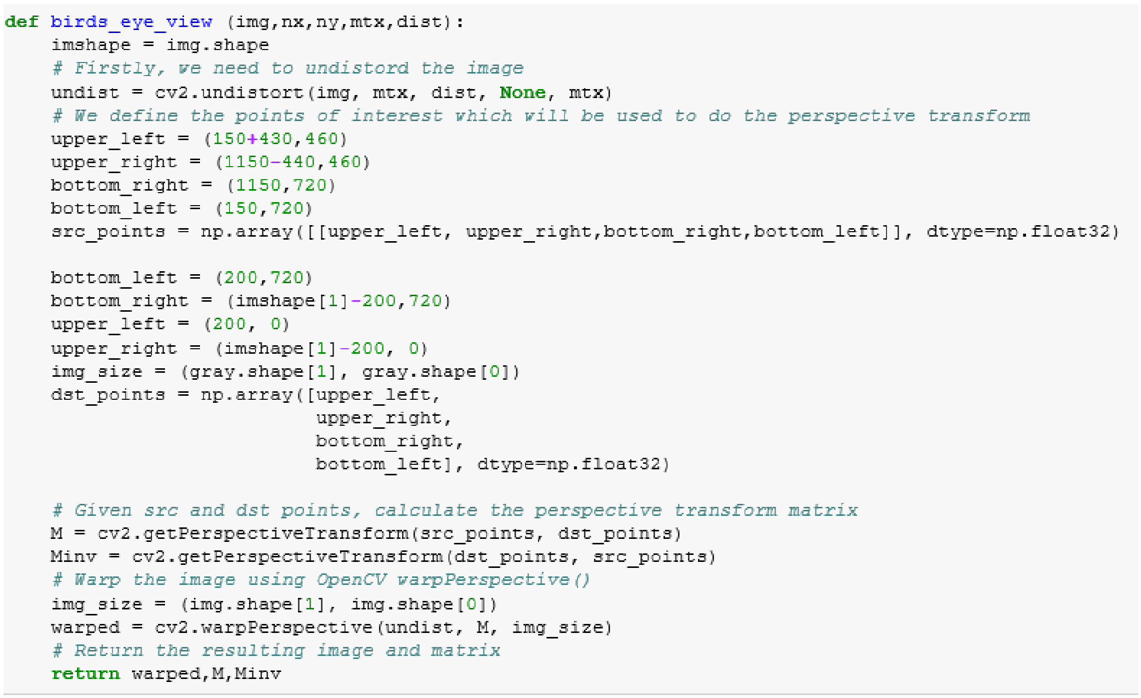
After making all the pipeline, I realized that there were still many things which were not required for detecting the lanes and could make my algorithm work wrongly. So, I decided to apply a ROI to the output of the pipeline. And this was the result:



2.3. Perspective Transformation

One of the most important parts of the project was to generate a “birds eye view” perspective which is required to analyse each frame and extract the correct information that we want to develop algorithms like Lane Keeping.

The most challenging part of this part of the project was to find the best location of the points which we want to transform.



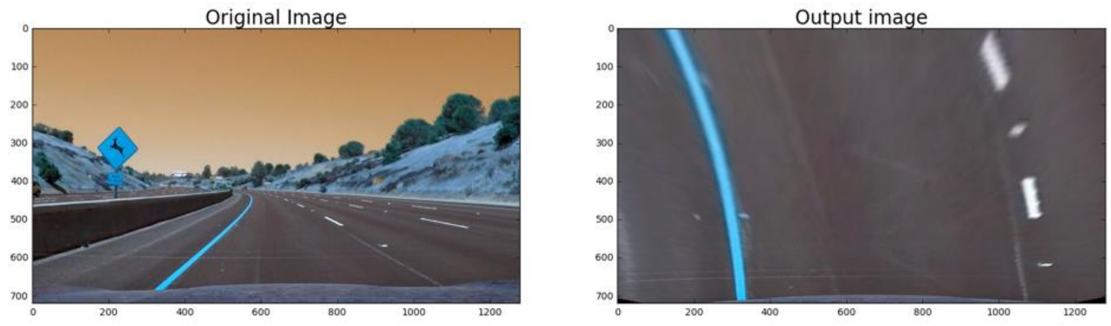
```python
def birds_eye_view (img,nx,ny,mtx,dist):
    imshape = img.shape
    # Firstly, we need to undistord the image
    undist = cv2.undistort(img, mtx, dist, None, mtx)
    # We define the points of interest which will be used to do the perspective transform
    upper_left = (150+430,460)
    upper_right = (1150-440,460)
    bottom_right = (1150,720)
    bottom_left = (150,720)
    src_points = np.array([[upper_left, upper_right,bottom_right,bottom_left]], dtype=np.float32)

    bottom_left = (200,720)
    bottom_right = (imshape[1]-200,720)
    upper_left = (200, 0)
    upper_right = (imshape[1]-200, 0)
    img_size = (gray.shape[1], gray.shape[0])
    dst_points = np.array([upper_left,
                           upper_right,
                           bottom_right,
                           bottom_left], dtype=np.float32)

    # Given src and dst points, calculate the perspective transform matrix
    M = cv2.getPerspectiveTransform(src_points, dst_points)
    Minv = cv2.getPerspectiveTransform(dst_points, src_points)
    # Warp the image using OpenCV warpPerspective()
    img_size = (img.shape[1], img.shape[0])
    warped = cv2.warpPerspective(undist, M, img_size)
    # Return the resulting image and matrix
    return warped,M,Minv
```

As it can be seen in the image before, I used the Open CV for make this perspective transformation and get the warped image and also the perspective matrix which will be required later one for do the reverse transformation from the detected lanes to the image.

The results of this function are shown in the following image:



2.4. Lanes Detection

Once, I was able to obtain a binary image which was useful for detecting the lanes, it was necessary to develop the functions which were required for detecting the position and curvature of such lanes.

2.4.1 Detecting lane pixels

In order to obtain the coefficients of the second order polynomial, it was required to obtain the position of the pixels. The approach that I followed was to firstly, extract the coordinates of the binary image which were 1 because are the only candidates to be part of the line.

This is the function generated for this purpose:

```python
def lanes_detector (n, image, x_window, lanes, \
                    left_lane_x, left_lane_y, right_lane_x, right_lane_y, window_ind):
    # 'n' windows will be used to identify peaks of histograms
    # Set index1. This is used for placeholder.
    index1 = np.zeros((n+1,2))
    index1[0] = [300, 1100]
    index1[1] = [300, 1100]
    # Set the first left and right values
    left, right = (300, 1100)
    # Set the center
    center = 700
    # Set the previous center
    center_pre = center
    # Set the direction
    direction = 0
    for i in range(n-1):
        # set the window range.
        y_window_top = 720-720/n*(i+1)
        y_window_bottom = 720-720/n*i
        # If left and right lanes are detected from the previous image
        if (left_lane.detected==False) and (right_lane.detected==False):
            # Find the historgram from the image inside the window
            left  = peaks_detector(image, y_window_top, y_window_bottom, index1[i+1,0]-200, index1[i+1,0]+200)
            right = peaks_detector(image, y_window_top, y_window_bottom, index1[i+1,1]-200, index1[i+1,1]+200)
            # Set the direction
            left  = lane_checks_direction(left, index1[i+1,0], index1[i,0])
            right = lane_checks_direction(right, index1[i+1,1], index1[i,1])
            # Set the center
            center_pre = center
            center = (left + right)/2
            direction = center - center_pre
        # If both lanes were detected in the previous image
        # Set them equal to the previous one
        else:
            left  = left_lane.windows[window_ind, i]
            right = right_lane.windows[window_ind, i]
        # Make sure the distance between left and right laens are wide enough
        if abs(left-right) > 600:
            # Append coordinates to the left lane arrays
            left_lane_array = lanes[(lanes[:,1]>=left-x_window) & (lanes[:,1]<left+x_window) &
                                    (lanes[:,0]<=y_window_bottom) & (lanes[:,0]>=y_window_top)]
            left_lane_x += left_lane_array[:,1].flatten().tolist()
            left_lane_y += left_lane_array[:,0].flatten().tolist()
            if not math.isnan(np.mean(left_lane_array[:,1])):
                left_lane.windows[window_ind, i] = np.mean(left_lane_array[:,1])
                index1[i+2,0] = np.mean(left_lane_array[:,1])
            else:
                index1[i+2,0] = index1[i+1,0] + direction
                left_lane.windows[window_ind, i] = index1[i+2,0]
            # Append coordinates to the right lane arrays
            right_lane_array = lanes[(lanes[:,1]>=right-x_window) & (lanes[:,1]<right+x_window) &
                                     (lanes[:,0]<y_window_bottom) & (lanes[:,0]>=y_window_top)]
            right_lane_x += right_lane_array[:,1].flatten().tolist()
            right_lane_y += right_lane_array[:,0].flatten().tolist()
            if not math.isnan(np.mean(right_lane_array[:,1])):
                right_lane.windows[window_ind, i] = np.mean(right_lane_array[:,1])
                index1[i+2,1] = np.mean(right_lane_array[:,1])
            else:
                index1[i+2,1] = index1[i+1,1] + direction
                right_lane.windows[window_ind, i] = index1[i+2,1]
    return left_lane_x, left_lane_y, right_lane_x, right_lane_y
```

In two possible ways will work this function:

- **When no lane was detected before:**

The approach in this case will be to use the histogram procedure in order to find the positions of the beginning of the lanes, and this will be done for each of the windows in the image.

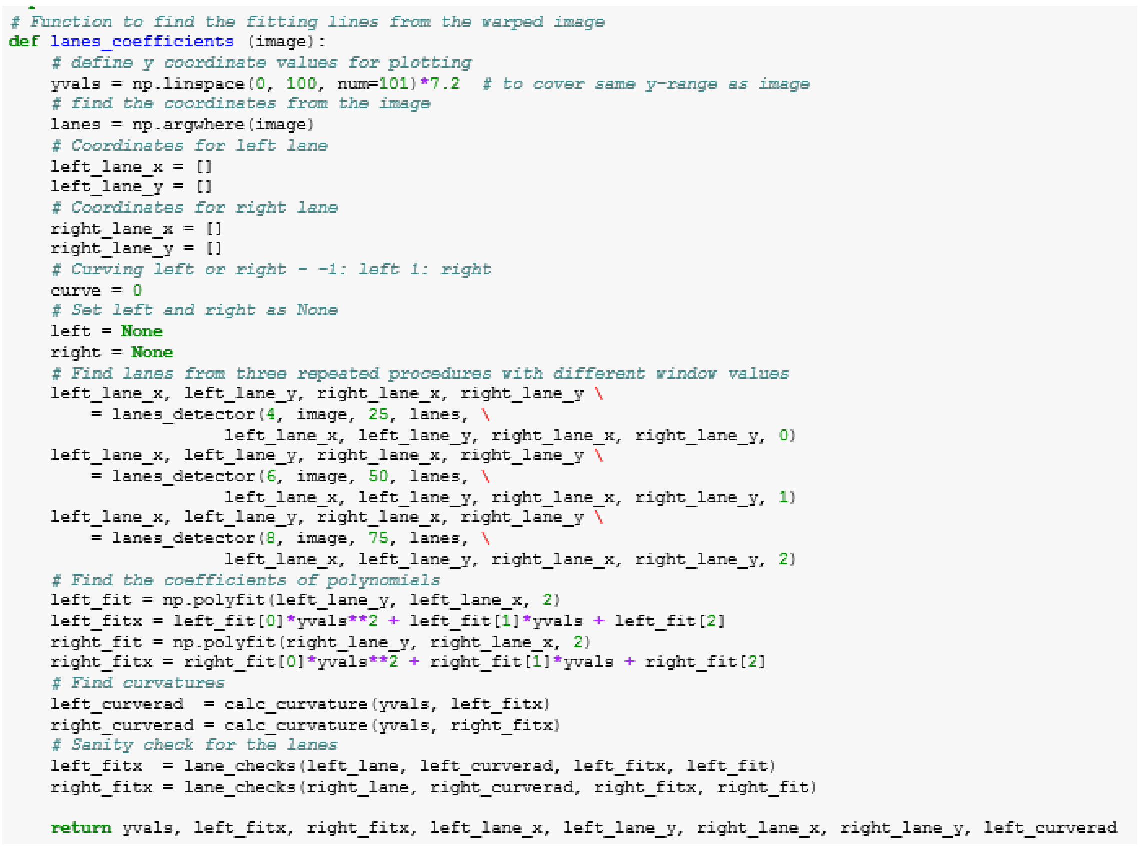
- **When a lane was detected before:**

In that case, the mean positions of the lane before will be used in order to find the pixel's candidates in the current image (it won't be necessary to calculate the histograms).

At the end, this function is returning the pixel's positions of the lanes right and left.

2.4.2. Calculating the lane coefficients

Once, we get the pixel's positions, the next step is to use the polyfit function from Open CV for calculating the coefficients of the lines.



```
# Function to find the fitting lines from the warped image
def lanes_coefficients (image):
    # define y coordinate values for plotting
    yvals = np.linspace(0, 100, num=101)*7.2  # to cover same y-range as image
    # find the coordinates from the image
    lanes = np.argwhere(image)
    # Coordinates for left lane
    left_lane_x = []
    left_lane_y = []
    # Coordinates for right lane
    right_lane_x = []
    right_lane_y = []
    # Curving left or right - -1: left 1: right
    curve = 0
    # Set left and right as None
    left = None
    right = None
    # Find lanes from three repeated procedures with different window values
    left_lane_x, left_lane_y, right_lane_x, right_lane_y \
        = lanes_detector(4, image, 25, lanes, \
                     left_lane_x, left_lane_y, right_lane_x, right_lane_y, 0)
    left_lane_x, left_lane_y, right_lane_x, right_lane_y \
        = lanes_detector(6, image, 50, lanes, \
                     left_lane_x, left_lane_y, right_lane_x, right_lane_y, 1)
    left_lane_x, left_lane_y, right_lane_x, right_lane_y \
        = lanes_detector(8, image, 75, lanes, \
                     left_lane_x, left_lane_y, right_lane_x, right_lane_y, 2)
    # Find the coefficients of polynomials
    left_fit = np.polyfit(left_lane_y, left_lane_x, 2)
    left_fitx = left_fit[0]*yvals**2 + left_fit[1]*yvals + left_fit[2]
    right_fit = np.polyfit(right_lane_y, right_lane_x, 2)
    right_fitx = right_fit[0]*yvals**2 + right_fit[1]*yvals + right_fit[2]
    # Find curvatures
    left_curverad  = calc_curvature(yvals, left_fitx)
    right_curverad = calc_curvature(yvals, right_fitx)
    # Sanity check for the lanes
    left_fitx  = lane_checks(left_lane, left_curverad, left_fitx, left_fit)
    right_fitx = lane_checks(right_lane, right_curverad, right_fitx, right_fit)

    return yvals, left_fitx, right_fitx, left_lane_x, left_lane_y, right_lane_x, right_lane_y, left_curverad
```

As it's really difficult to find the best parameters of the windows which are "analysing" each image frame, three times ***"lanes_detector"*** function has been called and all the coordinates have been saved in the same arrays.

Once the lanes pixels are detected the coefficients of the second order polynomial are found by using the Python function:

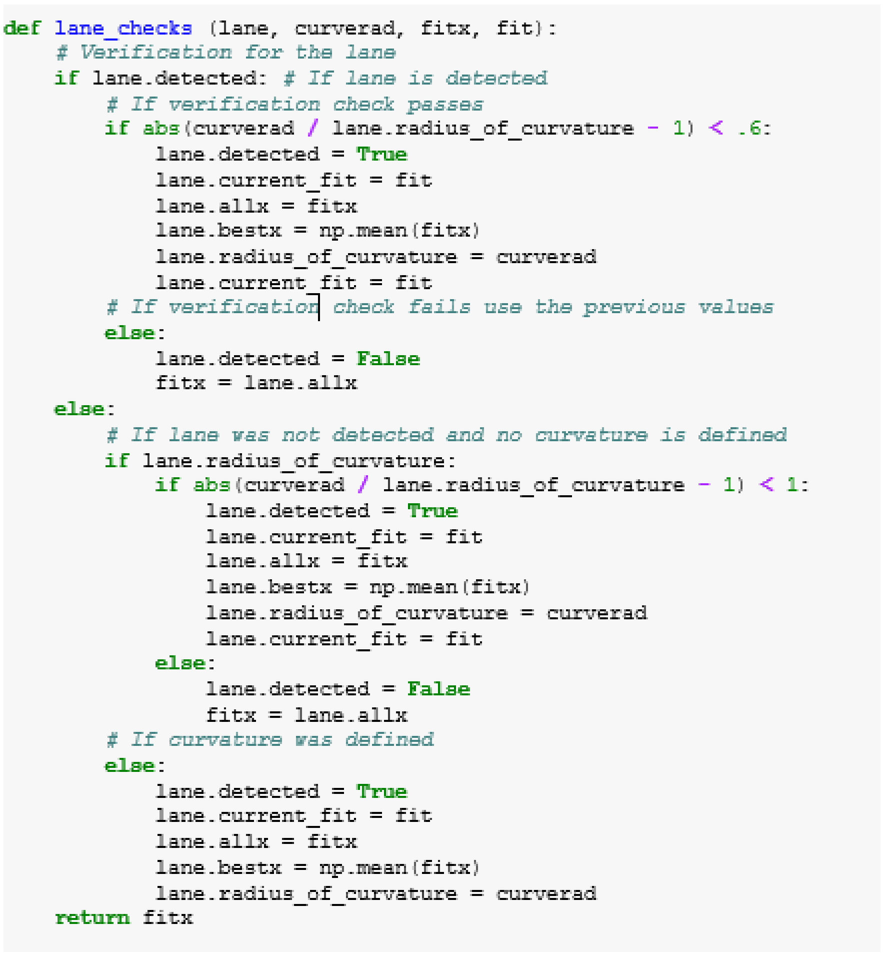
"np. Polyfit(lane_y, lane_x,2)".

Inside this functions, the radius of curvature is calculated and last step is to do some checks with the information of the lanes and curvature in order to avoid bad detection of lanes. This is calculated in the ***"lane_checks()"*** function.

2.4.2.1 Lane checks

The main objective of this function is to do some checks in order to not do wrong detection of the lanes.

That's the code of the function:



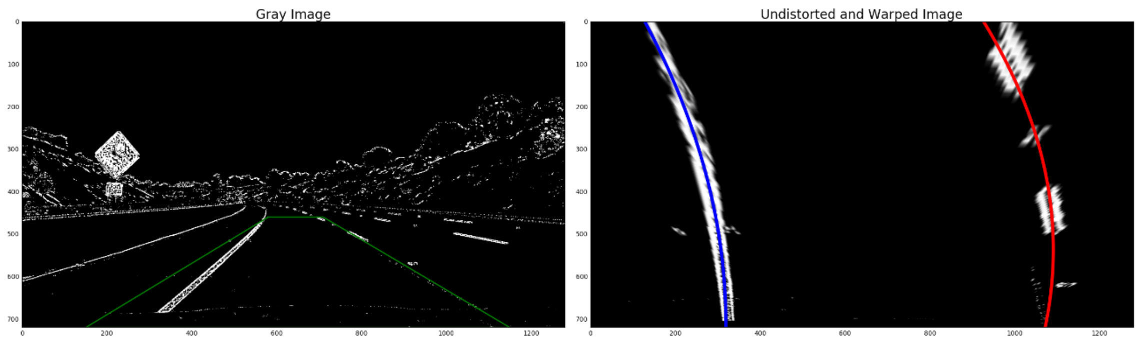
```python
def lane_checks (lane, curverad, fitx, fit):
    # Verification for the lane
    if lane.detected: # If lane is detected
        # If verification check passes
        if abs(curverad / lane.radius_of_curvature - 1) < .6:
            lane.detected = True
            lane.current_fit = fit
            lane.allx = fitx
            lane.bestx = np.mean(fitx)
            lane.radius_of_curvature = curverad
            lane.current_fit = fit
        # If verification check fails use the previous values
        else:
            lane.detected = False
            fitx = lane.allx
    else:
        # If lane was not detected and no curvature is defined
        if lane.radius_of_curvature:
            if abs(curverad / lane.radius_of_curvature - 1) < 1:
                lane.detected = True
                lane.current_fit = fit
                lane.allx = fitx
                lane.bestx = np.mean(fitx)
                lane.radius_of_curvature = curverad
                lane.current_fit = fit
            else:
                lane.detected = False
                fitx = lane.allx
        # If curvature was defined
        else:
            lane.detected = True
            lane.current_fit = fit
            lane.allx = fitx
            lane.bestx = np.mean(fitx)
            lane.radius_of_curvature = curverad
    return fitx
```

So, the main purpose of the function is detecting first if a lane was detected before. Depending on this:

- *When the lane was detected:* It's verified that the variation of the radius is bellow a threshold because it can't change too much from frame to frame.
- *When the lane wasn't previously detected:* Depending if the radius of curvature below a threshold or if the radius was defined, the lane will be detected.

2.4.3. Result of the lanes detector

In the following image, it’s shown the result of the algorithm in one frame.



2.5. Adding detected lines to the raw image

Once the line lanes were detected, the next step was to plot it in the raw image. To do this it was necessary to use the inverse perspective matrix from the bird’s eye view function.

The function for plotting such lines is shown below:

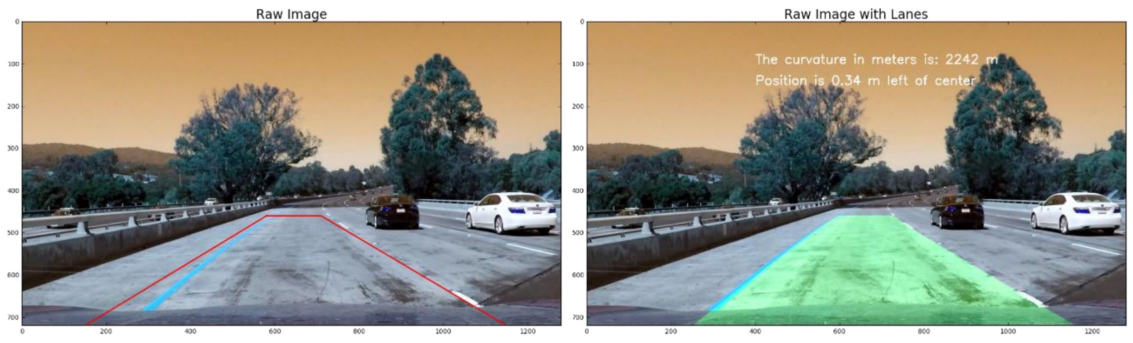
```python
def draw_poly(image, warped, yvals, left_fitx, right_fitx,
              left_lane_x, left_lane_y, right_lane_x, right_lane_y, Minv, curvature):
    # Create an image to draw the lines on
    warp_zero = np.zeros_like(warped).astype(np.uint8)
    color_warp = np.dstack((warp_zero, warp_zero, warp_zero))
    # Recast the x and y points into usable format for cv2.fillPoly()
    pts_left = np.array([np.transpose(np.vstack([left_fitx, yvals]))])
    pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, yvals])))])
    pts = np.hstack((pts_left, pts_right))
    # Draw the lane onto the warped blank image
    cv2.fillPoly(color_warp, np.int_([pts]), (0, 255, 0))
    # Warp the blank back to original image space using inverse perspective matrix (Minv)
    newwarp = cv2.warpPerspective(color_warp, Minv, (image.shape[1], image.shape[0]))
    # Combine the result with the original image
    result = cv2.addWeighted(image, 1, newwarp, 0.3, 0)
    # Put text on an image
    font = cv2.FONT_HERSHEY_SIMPLEX
    text = "The curvature in meters is: {} m".format(int(curvature))
    cv2.putText(result,text,(400,100), font, 1,(255,255,255),2)
    # Find the position of the car
    pts = np.argwhere(newwarp[:,:,1])
    position = calc_position(pts)
    if position < 0:
        text = "Position is {:.2f} m left of center".format(-position)
    else:
        text = "Position is {:.2f} m right of center".format(position)
    cv2.putText(result,text,(400,150), font, 1,(255,255,255),2)
    return result
```

As it’s shown what it’s done here is to use the raw image and the information of the line lanes and wrapped in the same output image.

To do this, it’s necessary to use the Open CV function ***“cv2.warpPerspective()”*** which takes the image with the lanes plotted in and the inverse perspective matrix and generates the transformed image to the camera perspective.

The las step is to just join the raw image with the one with the lanes plotted.

This is the result of such function:


Raw Image
Raw Image with Lanes
The curvature in meters is: 2242 m
Position is 0.34 m left of center

3. Conclusion

All the image results can be found in the folder “output_images” and also the resulting video of the project.

The most challenging part of the project has been to detect the lanes from the binary image because it was quite challenging to find the right way of making windows which where filtering each frame of the video and also to find the correct decisions for deciding if the detected lanes were wrong or fine.

However, the part which has to be improved to make this system work in all the situations in the real world is the part of generating the binary image. It’s quite important to make this algorithm work under any circumstances in the real world and this will complicate a lot the work for tuning the correct parameters or finding the best techniques to get the correct binary data.

Furthermore, if this has to be implemented in real-time it will be also necessary to optimize the code in order to meet the real-time requirements.

It has been amazing to learn how to use Open CV functions and python libraries in order to do these kind of algorithms. I’m quite surprised in how much you can do with this tools.