Advanced Lane Finding Project

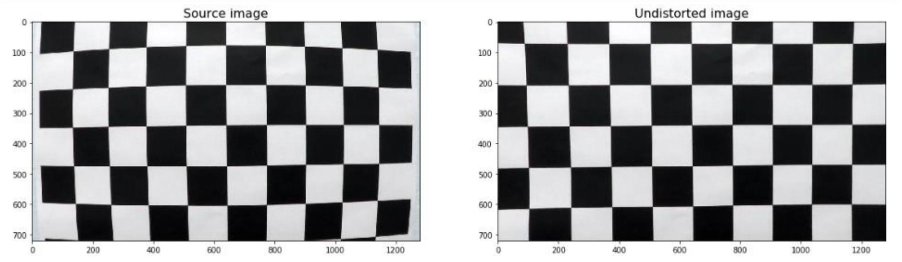
Camera Calibration:

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. The `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

Distortion-corrected image:

Using 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:

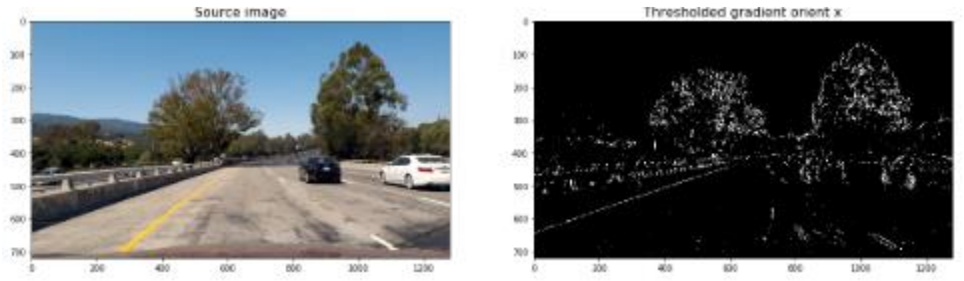
```
# Undistorting a sample image
img = cv2.imread("camera_cal/calibration1.jpg")
undistorted_img = cv2.undistort(img, mtx, dist, None, mtx)
plt_images(img, 'Source image', undistorted_img, 'Undistorted image')
```

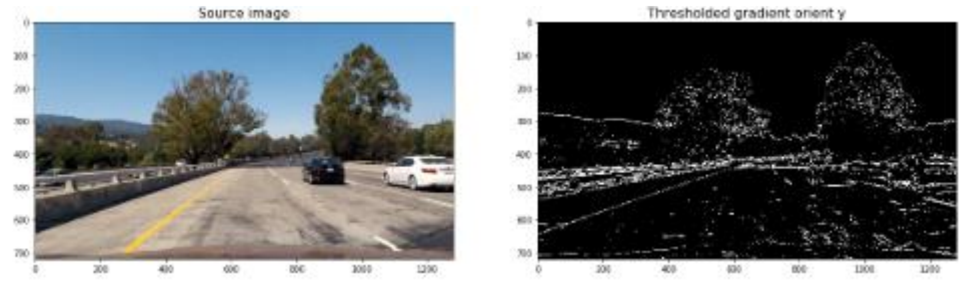


Color transforms, gradients or other methods to create a binary image:

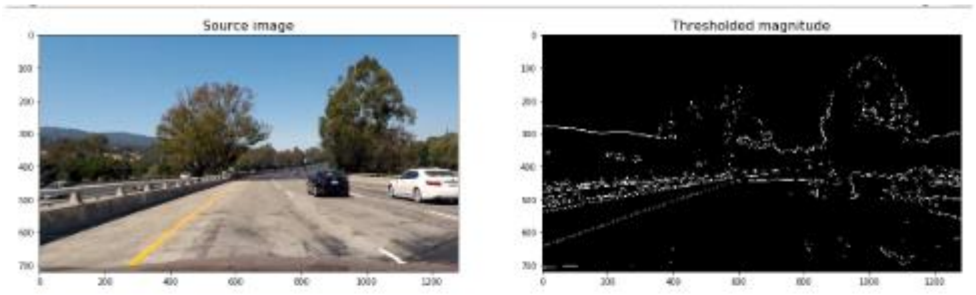
Here using the Sobel operator, I calculated the following and selected the best parameters and used combine_threshs function to identify the lanes clearly in the binary image.

Directional gradient

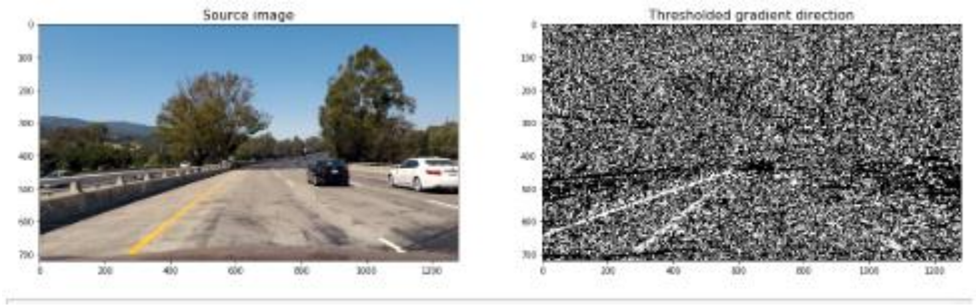




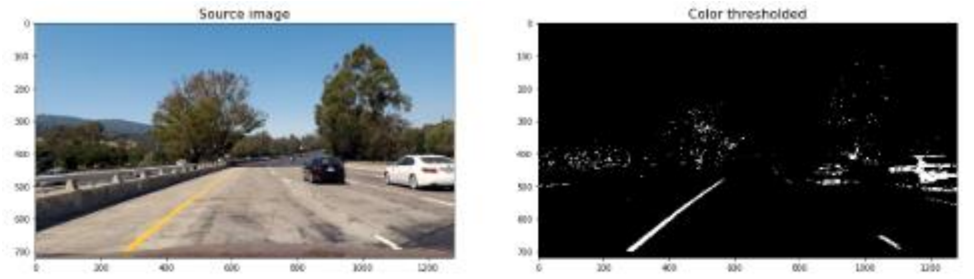
Gradients Magnitude



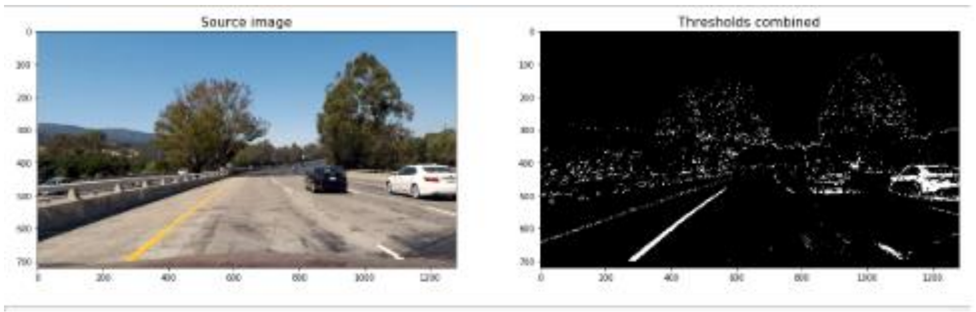
Gradient Direction



Color Threshold



Final Image:



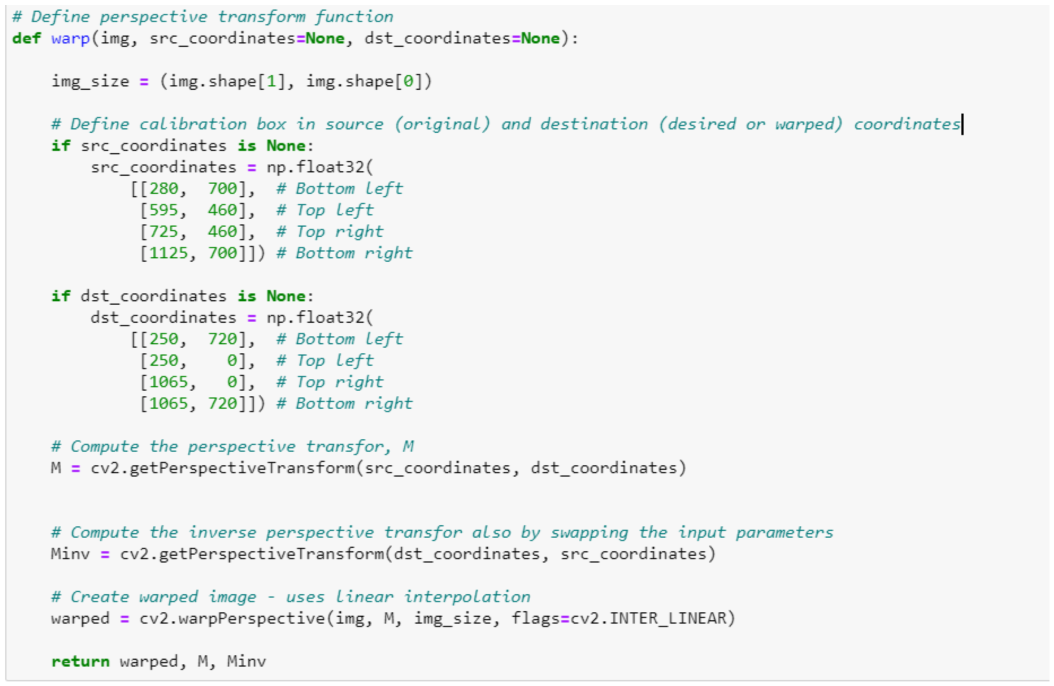
Perspective Transform To Warp and Unwarp The Image:

Steps:

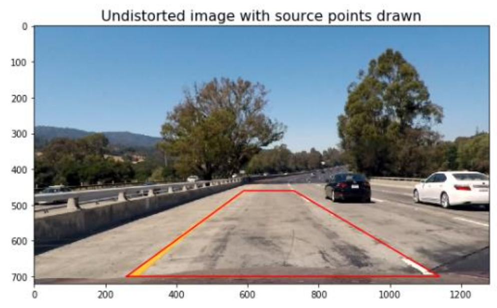
1.Select coordinates in the image enclosing the lanes in a trapezoidal shape

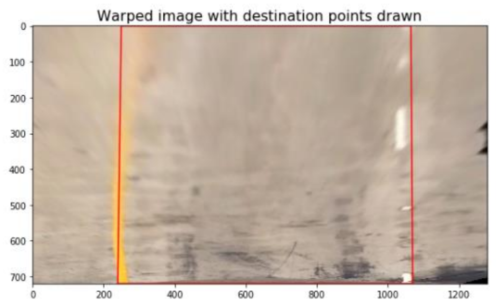
2.Apply the source and destination coordinates to cv2.getPerspectiveTransform() function and get both the perspective transform(M) and its inverse(Minv).

3.M and Minv will be used in the video to warp and unwarp it.



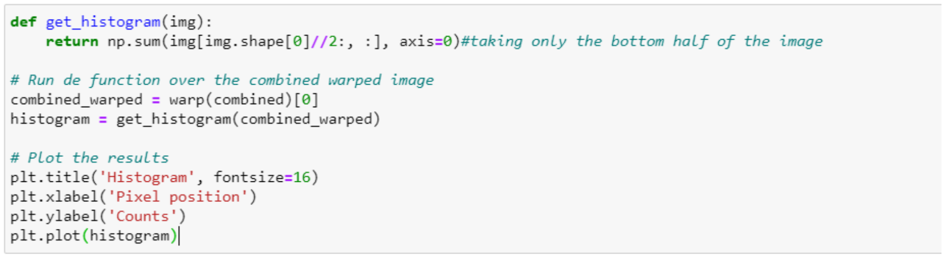
```python
# Define perspective transform function
def warp(img, src_coordinates=None, dst_coordinates=None):

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

    # Define calibration box in source (original) and destination (desired or warped) coordinates
    if src_coordinates is None:
        src_coordinates = np.float32(
            [[280,  700],  # Bottom left
             [595,  460],  # Top left
             [725,  460],  # Top right
             [1125, 700]]) # Bottom right

    if dst_coordinates is None:
        dst_coordinates = np.float32(
            [[250,  720],  # Bottom left
             [250,    0],  # Top left
             [1065,   0],  # Top right
             [1065, 720]]) # Bottom right

    # Compute the perspective transfor, M
    M = cv2.getPerspectiveTransform(src_coordinates, dst_coordinates)


    # Compute the inverse perspective transfor also by swapping the input parameters
    Minv = cv2.getPerspectiveTransform(dst_coordinates, src_coordinates)

    # Create warped image - uses linear interpolation
    warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)

    return warped, M, Minv
```



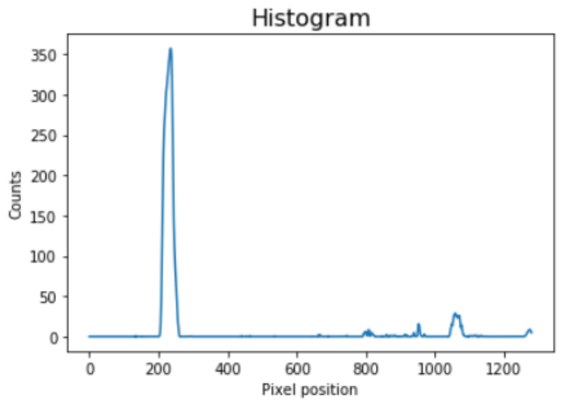


Identify the Peaks using Histogram to get the base coordinates of the left and right lane

Sliding Window concept was used to identify the most coordinates that will likely fall on the lane lines in a window, this will slide vertically through the image for both the lanes.



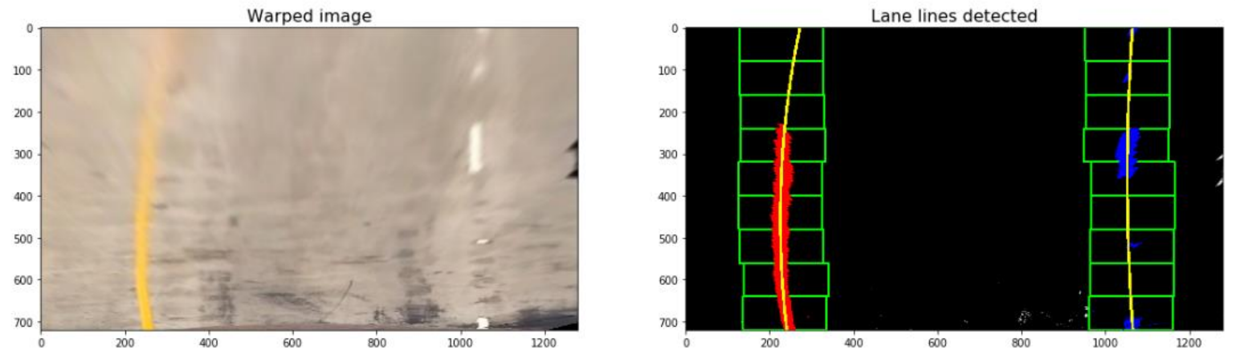
```
def get_histogram(img):
    return np.sum(img[img.shape[0]//2:, :], axis=0)#taking only the bottom half of the image

# Run de function over the combined warped image
combined_warped = warp(combined)[0]
histogram = get_histogram(combined_warped)

# Plot the results
plt.title('Histogram', fontsize=16)
plt.xlabel('Pixel position')
plt.ylabel('Counts')
plt.plot(histogram)
```



Using the coordinates previously calculated, a second-order polynomial is calculated for both the left and right lane lines. Once the lines are selected, it is reasonable to assume that the lines will have the same position video frames.

detect_similar_lines() uses the previously calculated line_fits to identify the lines consecutive images.

If it fails to calculate it, it calls the identify_lines () function to search for the lines again.

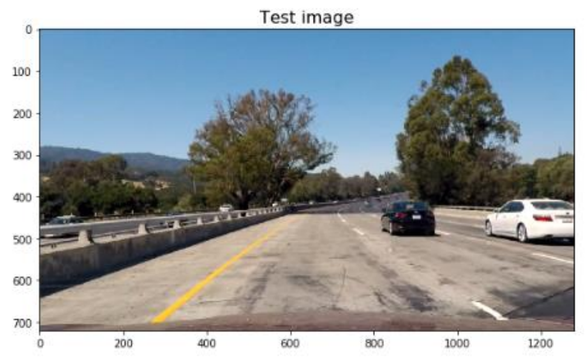


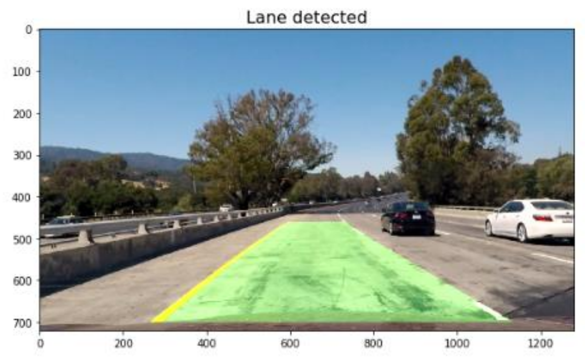
The curvature of the lines and the distance of the center of the car between the lanes is calculated

Warping the detected boundaries on the original image

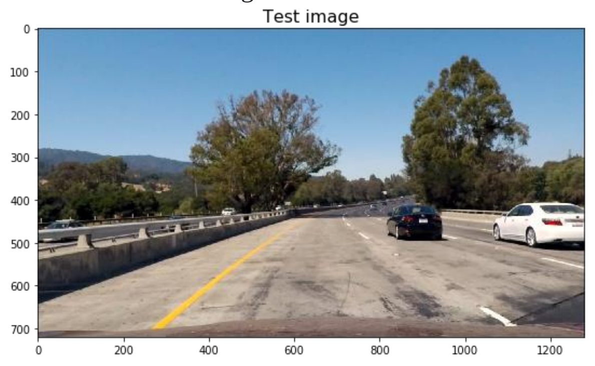
The lanes are drawn in the warped version of the blank image using cv2.fillPoly.

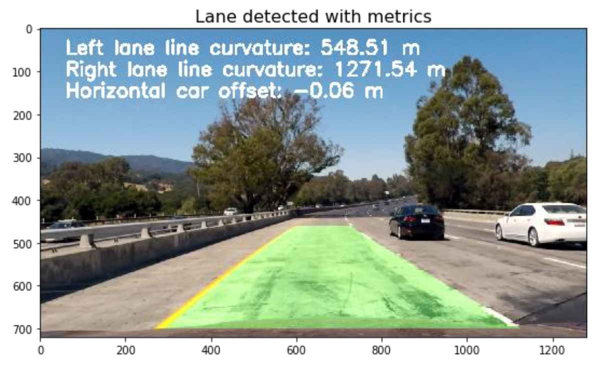
This is warped to the original image using the inverse perspective (Minv).





Displaying the calculated metrics along with the lanes
add_metrics() is used to perform this operation. It receives the image and the line points and returns an image that contains the left and right lane lines radius of curvature and the car offset.




Lane detected with metrics
Left lane line curvature: 548.51 m
Right lane line curvature: 1271.54 m
Horizontal car offset: -0.06 m

Pipeline:

The pipeline consists of all the above steps in sequence. I have tried using a class instead of a function to calibrate the camera when initializing the class and to keep a track of the previously detected lines.

Discussion

This was an extremely challenging project. It took me almost a week to complete.

I was able to come up with a good model to run the project video file.
Adjusting the gradients and color threshold manually was a bit tedious task.
The model is not accurate enough to detect the lanes when there are shadows.
Also, the model might fail if a white car is in proximity. Including a feature to identify and classify an object would be a possible solution.