Advanced Lane Lines

Fabrizio Frigeni, 2017.10.12

Specifications

The goal of this project is to recognize and highlight the current driving lane in a video taken by a vehicle front camera. Additionally, we need to calculate the current road curvature and the lateral offset of the vehicle from the center of the lane.

Implementation

All the required functions are implemented in the library file *mylib.py*. The program *main.py* is then used to read each frame of an existing video, process each frame with a pipeline of functions from mylib.py, and then save the frames back. The program *make_video.py* is finally used to stack all the processed frames into a new video.

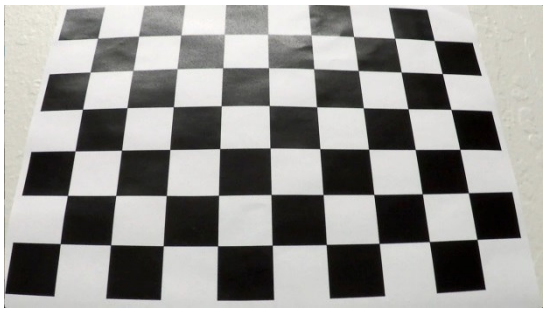
The following frame processing functions are described next:

- Camera calibration
- Thresholded binary image
- Perspective transform
- Line identification
- Geometrical calculations
- Inverse warping
- Moving average filter

Camera calibration

The purpose of camera calibration is to make sure that all images taken at real-time will appear free of distortions, so that object shapes will be retained correctly and image processing calculations will return correct results.

A simple way to calibrate a camera is to use a chessboard: it features high contrast edges, which are easy to find, and perfectly straight lines, which are essential to calibrate a 2D coordinate system. An example of distorted camera view is shown below. Notice the curved bottom edge of the chessboard.

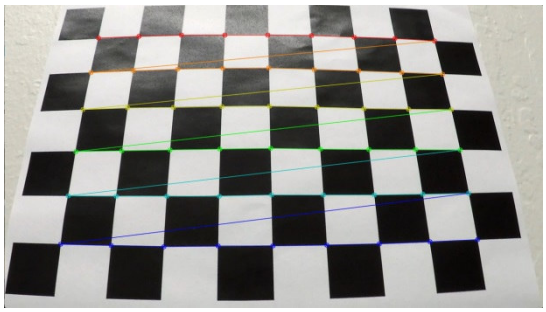


OpenCV offers built-in functions to calibrate a camera and undistort images.

We first need to take several images of the chessboard at different angles and locations. Then extract the calibration points from each image with the function *findChessboardCorners*:

```
# find chessboard corners
ret, corners = cv2.findChessboardCorners(gray, (9,6), None)
```

The 9x6 points of each image are automatically identified and stored in the array imgpoints:

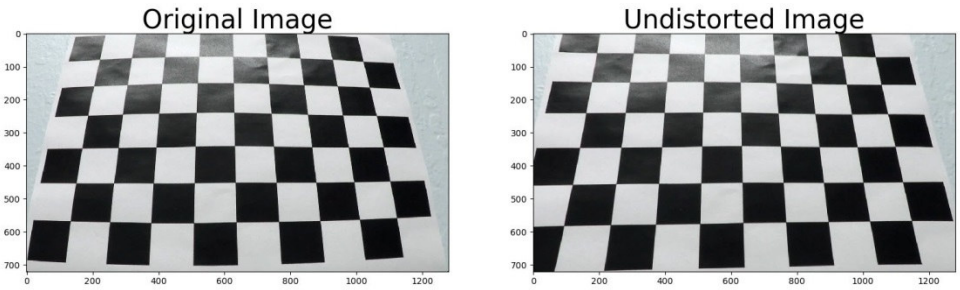


Next, we use the *calibrateCamera* function to calculate a transformation matrix *mtx* out of the calibration points:

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
img_size,None,None)
```

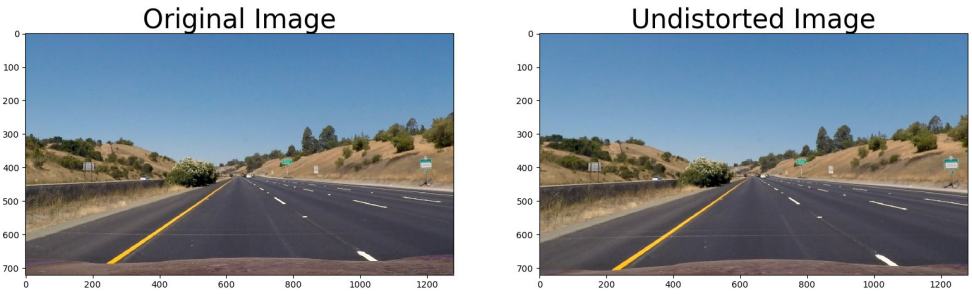
Applying the transformation matrix to a distorted image will produce an undistorted version, which we can then use for further processing:

```
dst = cv2.undistort(img, mtx, dist, None, mtx)
```



Notice how the curved bottom edge of the chessboard is transformed into a straight line.

Now we can apply the correction to real-time images:



The difference is minimal, mainly concentrated at the borders, but nevertheless essential for correct calculations in the following image processing steps.

Thresholded binary image

Once the image is free of distortions we can start working on it to find lane lines.

Lines usually have defined colors that contrast with the road background. They also appear at well-defined angles. Finally, they are only located in specific areas of the image. Using these properties we should be able to filter lines out of the background.

The *ImgToBinary* function was created for this purpose and applies several image processing techniques.

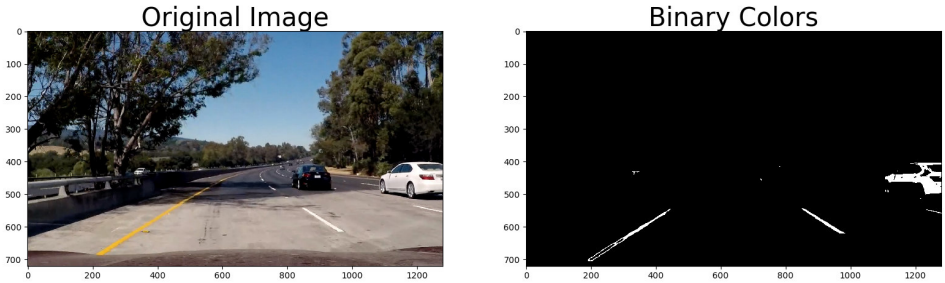
We first start with colors and analyze the image in the different spaces. Line colors per se are not important, e.g. could be either white or yellow, but their luminosity and saturations differentiate them from the background.

So we extract the H, L, S, and V channels from the image, filter them with thresholds and combine them together to increase robustness to different lighting conditions:

```
img_hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
img_hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
H = img_hls[:,:,0]
L = img_hls[:,:,1]
S = img_hls[:,:,2]
```

```
V = img_hsv[:,:,2]
binary_colors =  np.zeros_like(L)
binary_colors[((S > 50) & (S < 200))|((L > 200) & (L < 240))|((H > 200) & (H < 220))] = 1
binary_colors[V < 220] = 0
```

The last line is particularly important because it removes all areas that are too dark, which is extremely effective in removing all shaded areas of the road.



Then we turn our attention to edges. We work with gray raw pixels intensity gradients and analyze their absolute (normalized) value along each axis, their magnitude and their direction. Applying thresholds to each of these values we can filter out unwanted pixels and only concentrate on the ones that contain strong edges in the desired direction.

1. Convert image to gray space

```
gray = cv2.cvtColor(gray, cv2.COLOR_RGB2GRAY)
```

2. Calculate normalized gradients along x and y axes with Sobel filter

```
sobel_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
abs_sobel =  np.absolute(sobel_x)
scaled_sobel_x = np.uint8(255*abs_sobel/np.max(abs_sobel))
sobel_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1)
abs_sobel =  np.absolute(sobel_y)
scaled_sobel_y = np.uint8(255*abs_sobel/np.max(abs_sobel))
```

3. Calculate total magnitude of gradients

```
gradmag = np.sqrt(sobel_x**2 + sobel_y**2)
scale_factor = np.max(gradmag)/255
gradmag = (gradmag/scale_factor).astype(np.uint8)
```

4. Calculate direction of gradients

```
absgraddir = np.arctan2(np.absolute(sobel_y), np.absolute(sobel_x))
```

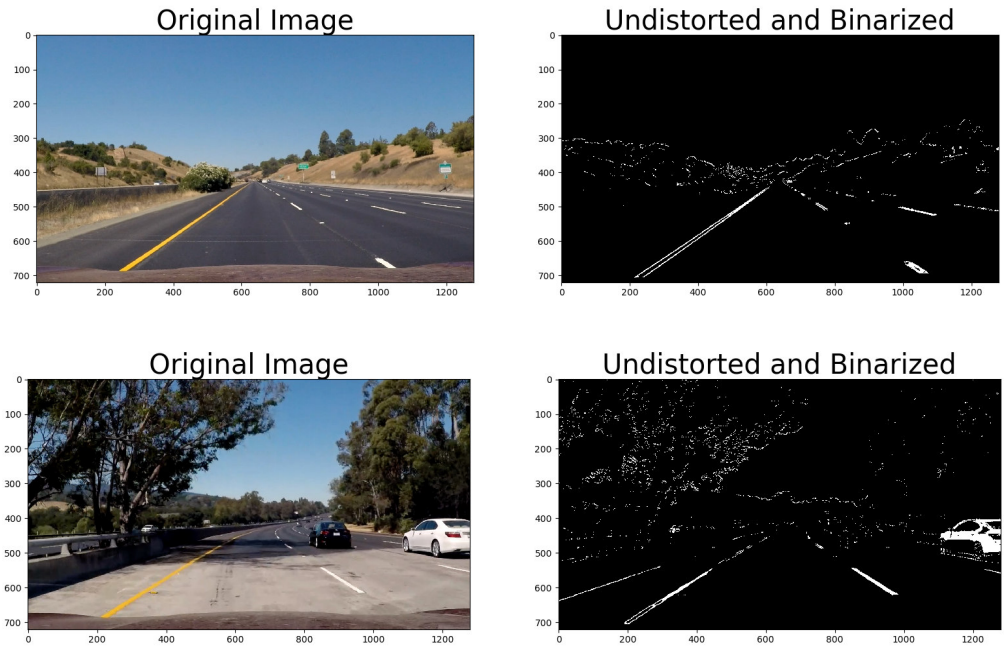
5. Apply thresholds to results of 2,3,4 and filter out pixels that fall out of thresholds intervals

```
binary_gradients = np.zeros_like(gradmag)
binary_gradients [(gradmag >= 30) & (gradmag <= 255)
                & (absgraddir >= 0.7) & (absgraddir <= 2)
                & (scaled_sobel_x >= 30) & (scaled_sobel_x <= 255)
                & (scaled_sobel_y >= 30) & (scaled_sobel_y <= 255)] = 1
```

Finally, we combine the two techniques (colors and gradients) into a single binary image:

```
combined_binary = np.zeros_like(binary_gradients)
combined_binary[(binary_colors == 1) | (binary_gradients == 1)] = 1
```

The result is a binarized version (i.e. each pixel can be either 0 or 1) of the original image:



The lane lines are clearly highlighted and separated from the road background.

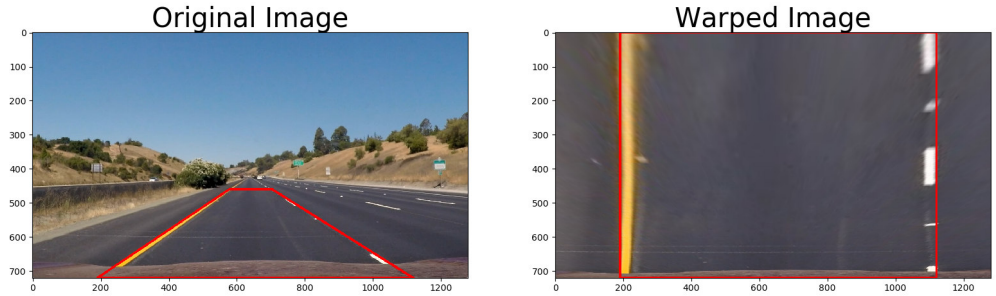
Perspective transform

Now that we have highlighted lane lines in the original image we need to look at their shape to measure their direction and curvature. However, when seen from the current front drive perspective the lines do not look parallel to each other and it is difficult to measure their correct curvature.

The solution is to look at them from a different perspective, namely from above. A bird's eye view shows parallel lines and makes the curvature calculation much easier.

OpenCV provides the function *warpPerspective* to change the perspective of an image given a transformation matrix. The matrix "M" can be found using the function *getPerspectiveTransform* given a set of source and destination points in the image. Its inverse "Minv" can be calculated analogously.

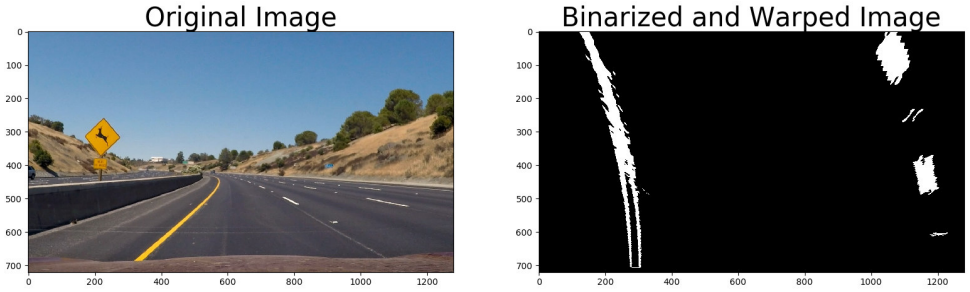
We combined the two previous built-in functions in the wrapper user function *ImgUnwarp*, where all the needed points are hard coded for simplicity. All images we deal with come from the same camera and the transformation matrix will not change over time (assuming a flat road for now).



Notice how the lines appear parallel when seen from above.

Line identification

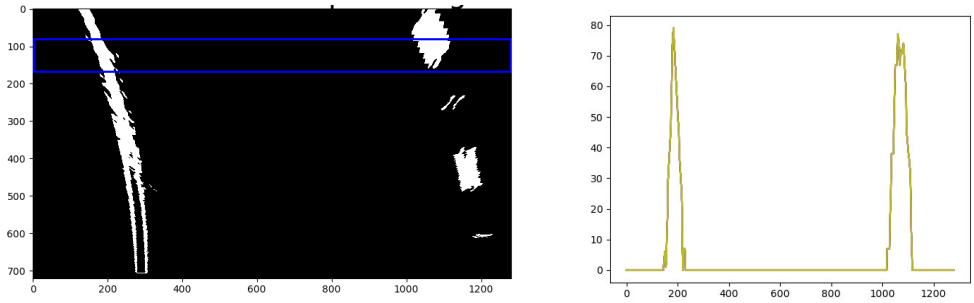
We are now able to combine the previous results and have a top view of the lines separated from the background. Even for curved road the lines still appear parallel to each other:



The goal is now to identify what pixels of the image belong to the two lines and fit a polynomial curve through them.

We can simply slice the image with several horizontal windows and recognize the lines by looking at the peaks of the intensity histogram. For example, the blue window on the left generates the histogram on the right:

```
histogram = np.sum(img[lowY:highY,:], axis=0)
```



Two peaks are clearly visible: we can now select their maximum values as the average horizontal position of the lines in that window.

```
middle = np.int(histogram.shape[0]/2)
left_peak_intensity = np.max(histogram[:middle])
right_peak_intensity = np.max(histogram[middle:])
left_peak = np.argmax(histogram[:middle])
right_peak = np.argmax(histogram[middle:]) + middle
```

However, during a video analysis we can also build in some additional information, considering that we already know where the lines are in the previous frame. So the search for histogram peaks can be concentrated in a smaller window (of fixed size *delta*) around the previous peaks (*prev_left_pos* and *prev_right_pos*), thereby escaping possible errors of false positives:

```
left_peak_intensity = np.max(histogram[prev_left_pos-delta:prev_left_pos+delta])
left_peak = np.argmax(histogram[prev_left_pos-delta:prev_left_pos+delta]) +
(prev_left_pos-delta)

right_peak_intensity = np.max(histogram[prev_right_pos-delta:prev_right_pos+delta])
right_peak = np.argmax(histogram[prev_right_pos-delta:prev_right_pos+delta]) +
(prev_right_pos-delta)
```

Sliding the window vertically across the whole image generates a set of points for the left line and another set for the right line. We then fit a quadratic curve through the points and obtain the quadratic coefficients to approximate the lane lines.

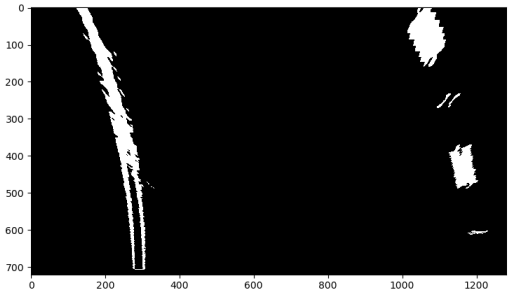
```
left_fit = np.polyfit(PointsLeftY, PointsLeftX, 2)
right_fit = np.polyfit(PointsRightY, PointsRightX, 2)
```

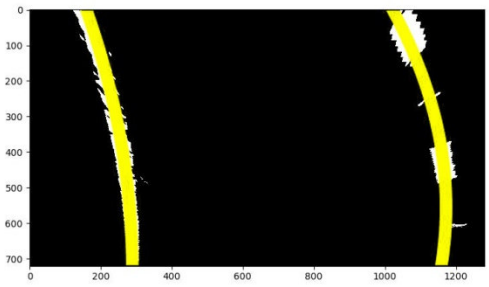
To make the points detection a bit more robust we add points to the array only if the peak intensity is larger than a minimal threshold, so to avoid noise disturbances.

```
if (np.max(histogram[:middle]) > 1):
        PointsLeftX.append(left_peak)
```

We noticed that around 1.8% of the frames did not pass this condition for the entire set of windows, which means they were not able to collect enough points for a reasonable polynomial fit.

The whole procedure is implemented in the *FindLines* function. This is what a result from a test image looks like:





Geometrical calculations

Identifying the bounding lines allows us to find two important parameters for driving control:

1. The position of the car with respect to the center, which tells us if we are following the lane correctly.
2. The lane's radius of curvature, which gives an idea of how sharp the turn is.

The offset from the center is quite easy to find. Assuming that the camera is mounted in the middle of the car, we measure the distance between the center of the image and the middle points between the two lines:

```
offset = X/2 - (PointsRightX[0]+PointsLeftX[0])/2
offset *= x_scaling
```

A positive offset means that the car is on the right side of the lane, while a negative offset means that the car is veering left. Notice that we need to scale from pixels into meters for a correct reading.

The radius of curvature can be calculated from the fitting polynomials by using the formula provided in the instructor's notes:

$$R = \frac{(1 + f'^2)^{3/2}}{|f''|}$$

We used the notation f' and f'' for the first and second derivatives of the polynomial function. Since the functions we are dealing with are quadratic, their derivatives are trivial to find and the final expression for the radius of curvature is implemented as:

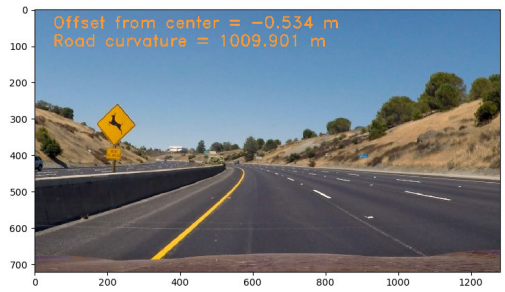
```
left_radius = ((1 + (2*left_fit_m[0]*y_eval*y_scaling + left_fit_m[1])**2)**1.5) / np.absolute(2*left_fit_m[0])
```

```
right_radius = ((1 + (2*right_fit_m[0]*y_eval*y_scaling + right_fit_m[1])**2)**1.5) / np.absolute(2*right_fit_m[0])
```

The lines are parallel and the two radii should be equal to each other. To reduce calculation errors we output an average value of the two sides:

```
radius = (left_radius+right_radius)/2
```

For the image below we get an offset of -0.5m, which means we are driving slightly left of the lane center, and a radius of curvature of about 1km, which corresponds well to the instructor's notes.



Inverse warping

The identified lines give us a good idea of the entire lane boundaries and we can easily plot them on the bird's eye view. However, we now need to transform it back into the original perspective from the car's front point of view.

In order to do that we first draw a filled polygon inside the two lines to cover the whole lane. We use the OpenCV function *fillPoly* for that purpose, by passing a set of points [pts] from the two lines stacked together:

```
cv2.fillPoly(warped_lane, np.int_([pts]), (0,255,0))
```

Then we unwarp the polygon back to the original front's view using the inverse perspective transformations matrix "Minv", which we had previously calculated in the *ImgUnwarp* function:

```
unwarped_lane = cv2.warpPerspective(warped_lane, Minv, (img.shape[1], img.shape[0]))
```

The un-warped lane now nicely fits into the original image and covers the lane area in front of the car:

Moving average filter

The polynomial fitting and curvature radius calculations are strongly dependent on the quality of the final image we obtain from the frame processing. Different light conditions and different shades of the road background can strongly affect out measurements, even if only for a few frames.

In order to reduce jumping between very different values and keep the representation smoother, we decided to add a filter to all calculated values: the polynomial coefficients, the curvature radius and the center offset. The function *MovingAverageFilter* was implemented with two simple rules:

- first, we filter out new values that are way off from the current buffer average, because they are likely caused by bad measurements and should not affect our general observations:

```
average = np.average(old_array)
diff = np.abs(new_value-average)
if diff > np.abs(average*threshold) and threshold > 0:
        return average, old_array
```

  We observed that about 0.8% of the frames were rejected by this sanity check.

- second, if a new value has passed the previous condition, we add it to the FIFO buffer and calculate the new average:

```
new_array = np.roll(old_array, 1)
new_array[0] = new_value
average = np.average(new_array)
return average, new_array
```

The size of the buffer is set to 5 values. The result is a smoother visualization of the lane area, especially when large portions of the side lines are missing. Increasing the buffer size makes the updates even smoother but somewhat too slow for fast changing curvature profiles.

We applied the entire processing pipeline to the *project_video.mp4* and the result was saved as *project_video_processed.mp4*. Note that the separate program *make_video.py* was used to convert the processed frames into video.

Comments

The currently implemented image processing pipeline is relative simple and only works for frames where lines are clearly visible. It is not robust enough to handle strong changes in lighting conditions and in road slopes.

One thing I would add is a dynamic selection of the ROI to detect lines depending on the current curvature, lateral offset and slope of the road. That should avoid false positive detections. The fixed ROI that we use at the moment fails badly when applied to tight curves.

Another major issue is the high computational cost of the current solution, especially with the sliding window approach in the line identification function, which makes the current solution unfeasible for real-time applications.