# Advanced Lane Finding

## Required Files

My project includes the following files / relevant folders:

- pipeline.ipynb is the primary file of this project and contains the full pipeline to generate the final output video
- output_images/ contains the individual images for each step of the pipeline
- examples.ipynb is a file that has been used to generate the outputs for this report
- calibration.py, threshold.py, transform.py and lane_detection.py are all files that encapsulate specific behaviour for the pipeline and referenced / imported in pipeline.ipynb
- output_video.mp4 is the final video output

## 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 the camera calibration is located in lines 8 through 49 of the file called `calibration.py` ).

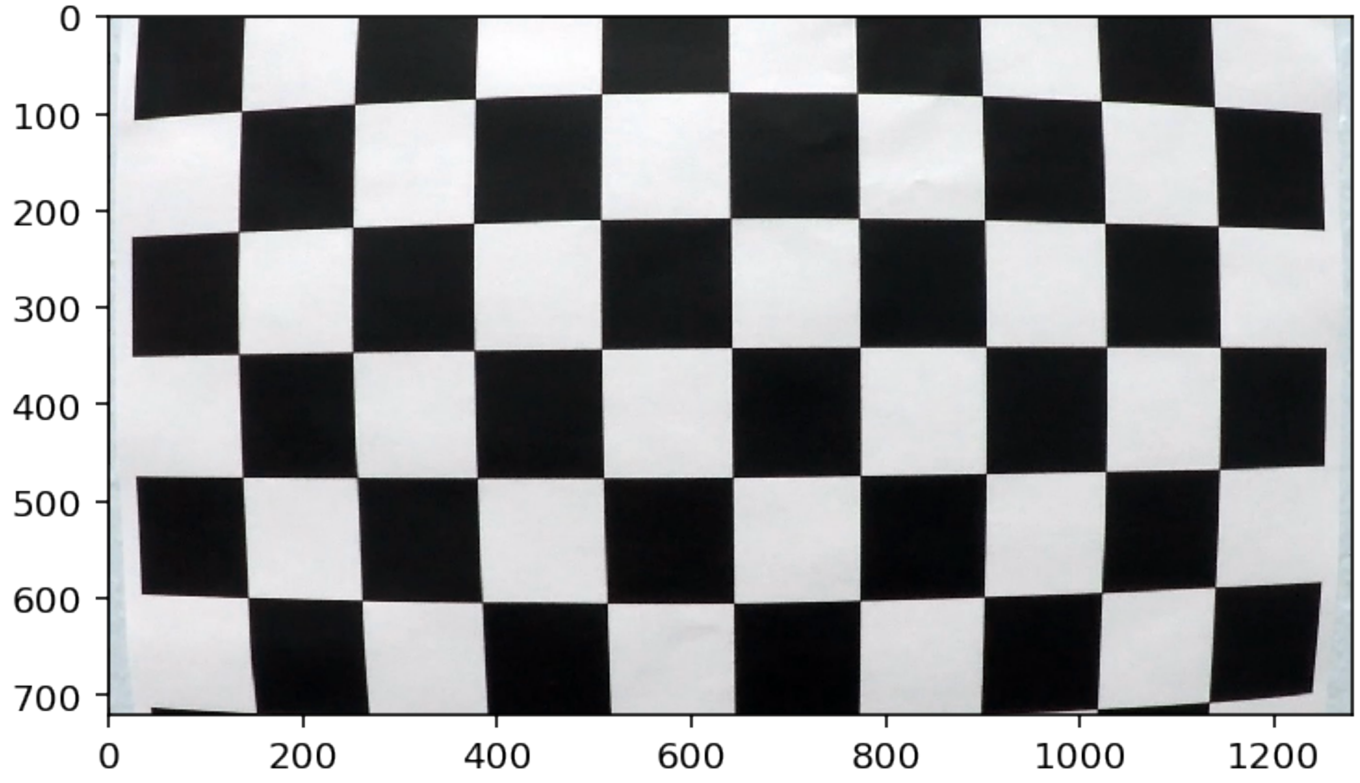
The function `calibrate` handles the camera calibration and return the camera matrix as well as the distortion coefficients. These can later on be used to undistort all frames of the video. When the function has run before, the pickled matrix and coefficients will not be recalculated but rather be unpickled and returned (code lines 11-14).

Apart from that, the function first generates the points in real world space (called object points) that describe the chessboard "in the real world". Points are thereby three-dimensional, whereas the z-axis is held constant (fixed to 0). The generation is done using numpy's `mgrid` function and a reshaping that outputs all x/y coordinates in object space (code lines 17-18).

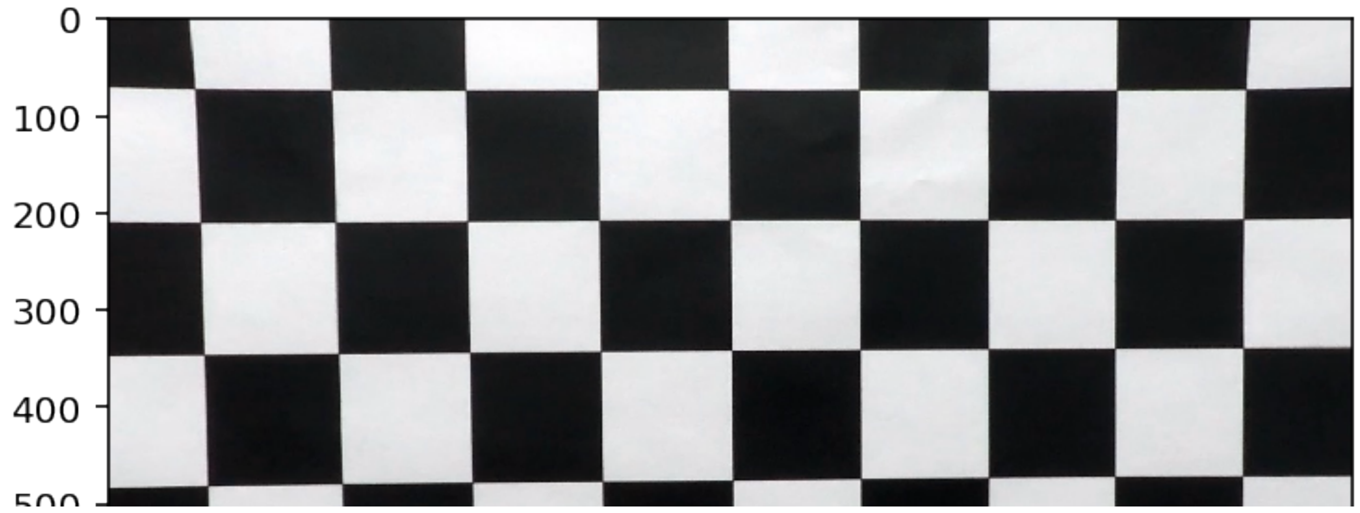
Afterwards all calibration images are loaded (code line 27), transformed into grayscale (code line 28), and image points are detected by OpenCV's `findChessboardCorners` function (code line 31). If all chessboard corners were found, the object points as well as the object points get collected in the lists `objpoints` and `imgpoints`.

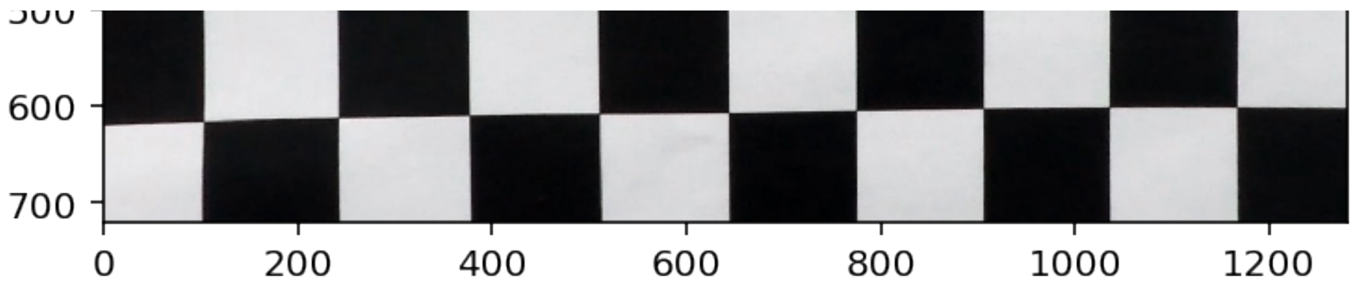
After all images have been processed, OpenCV's `calibrateCamera` function is called to actually calculate the camera matrix and the distortion coefficients.

An example image of a calibration image is this one:



And here the same image after undistorting it:



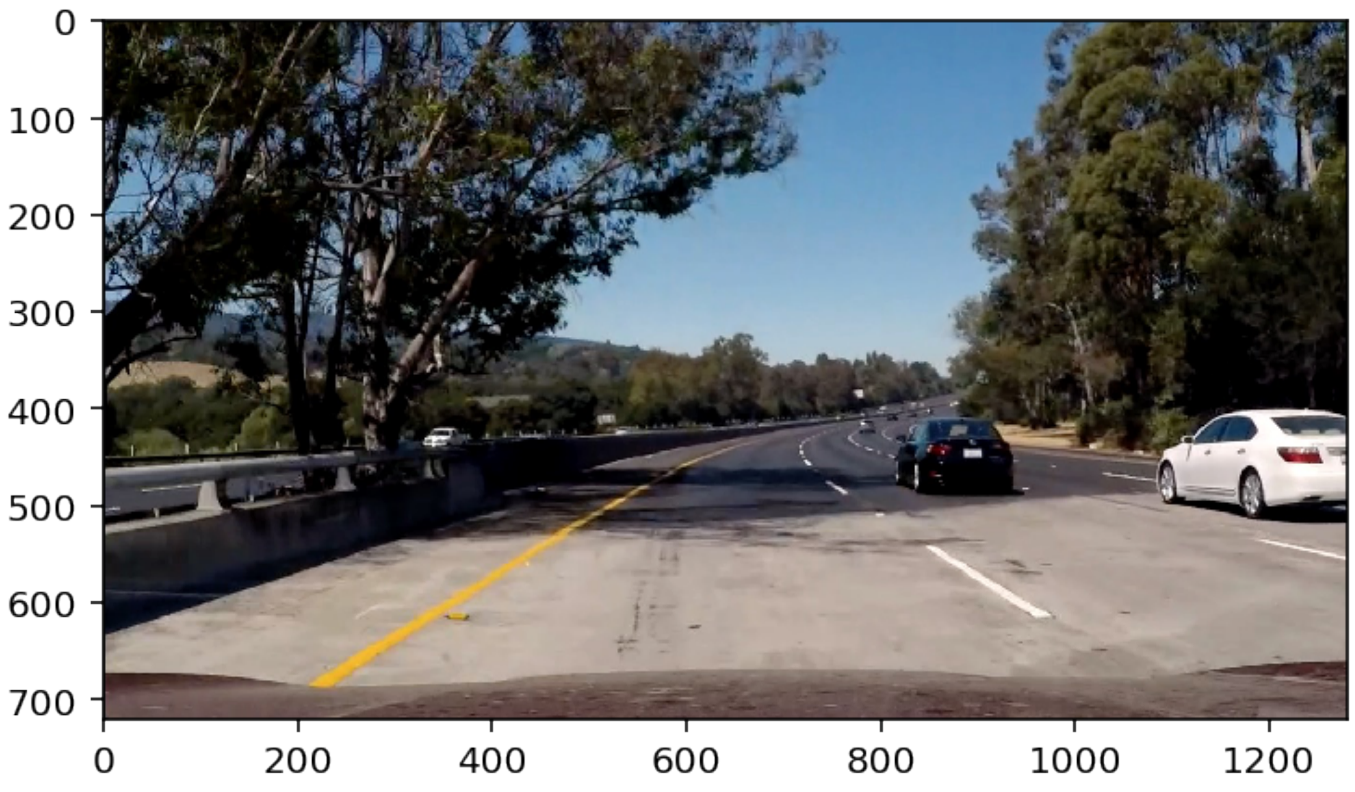


## Pipeline (single images)

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

The pipeline step for undistorting an input image can be found in code line 3 of the function `process_image` in the file `pipeline.ipynb`. For this purpose, the OpenCV function `undistort` is used (given the distortion coefficients and camera matrix. An example input and undistorted output look as follows:

Input:

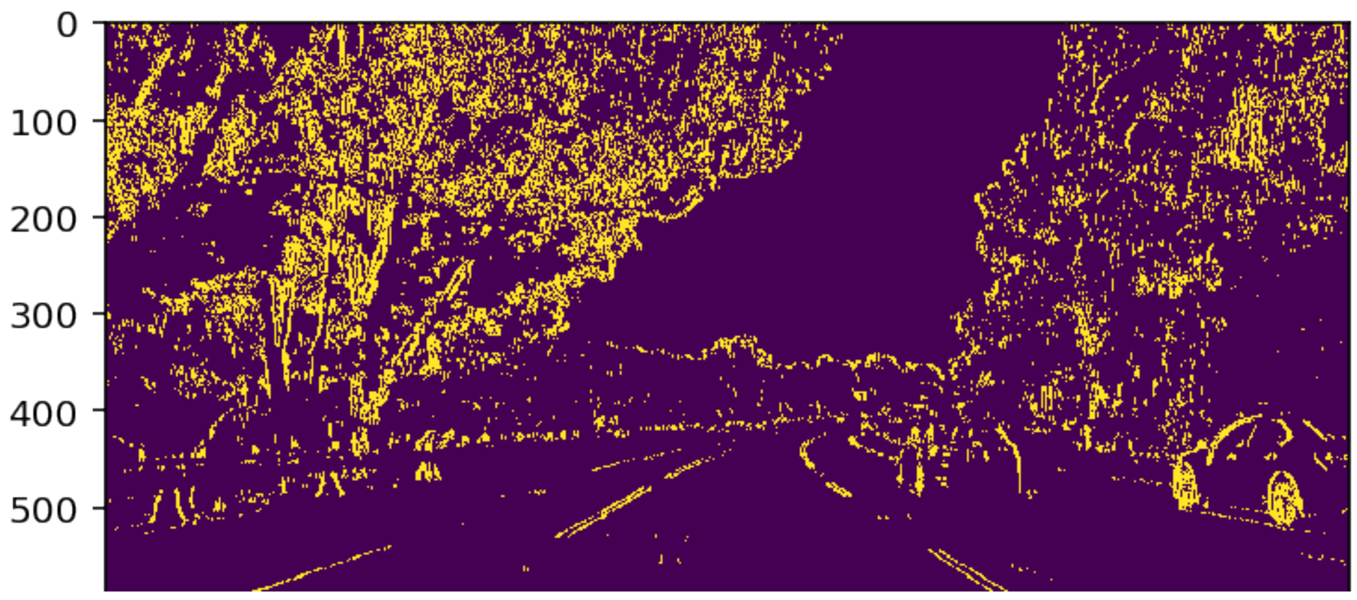


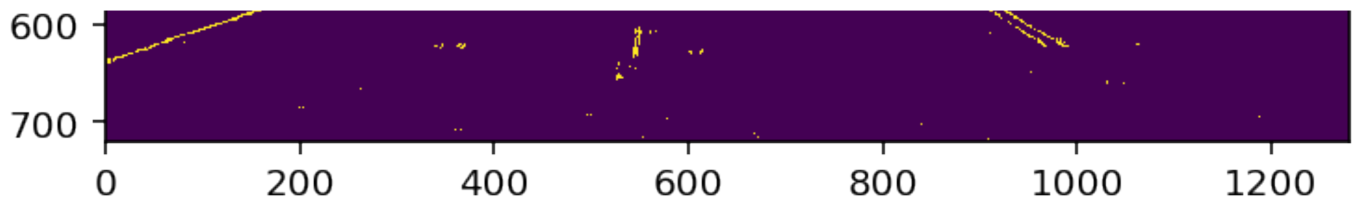
Output:

## 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.

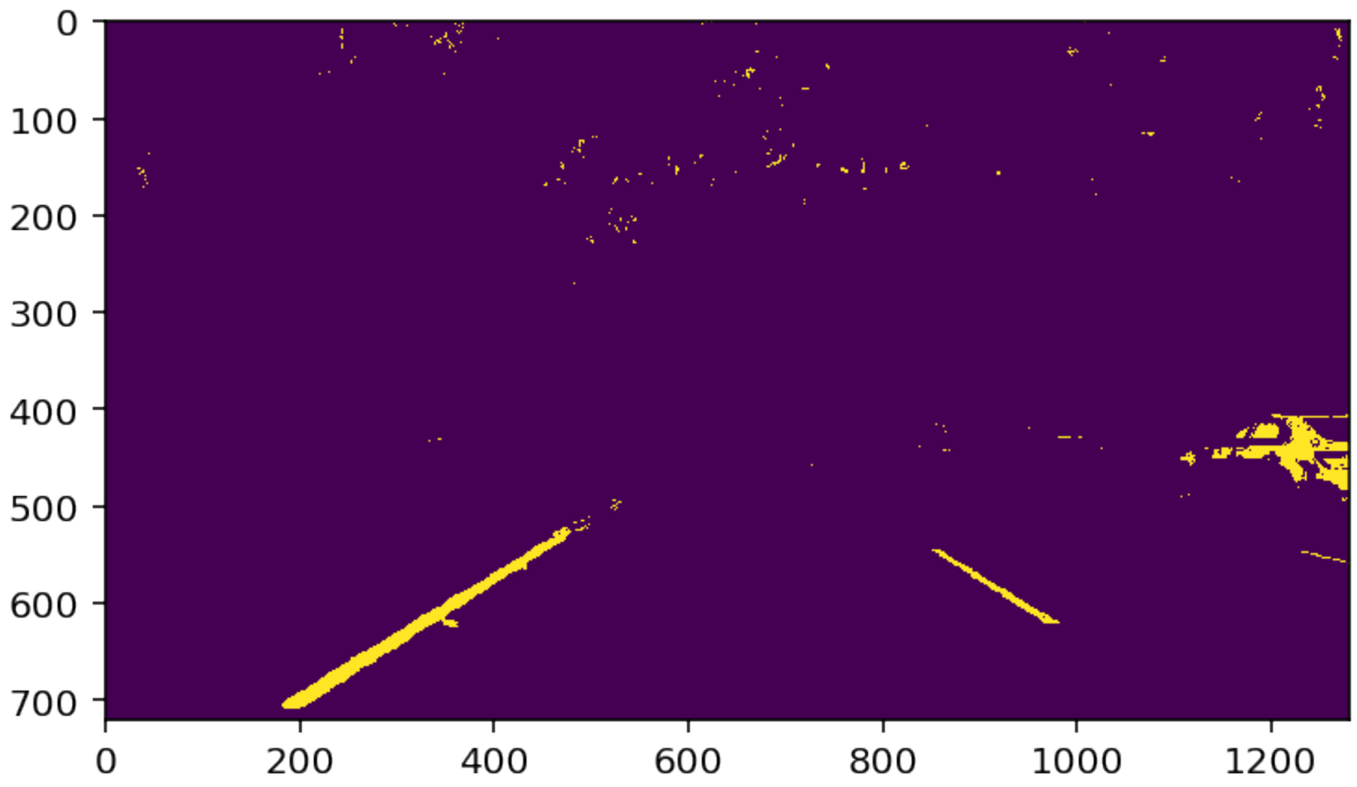
I ended up using a combination of gradient and color thresholding. Especially for the test images 4 and 5 I had difficulties in dealing with the shadows of the trees as well as the color differences of the road. I experimented with basically all the methods introduced in the lectures. However, eventually I chose a combination of an x gradient with a thresholds between [20, 255] and a HLS-converted image and thresholds on the L [120, 255] and S [20, 255] channels which gave me good results on all test frames as well as the video.

X:

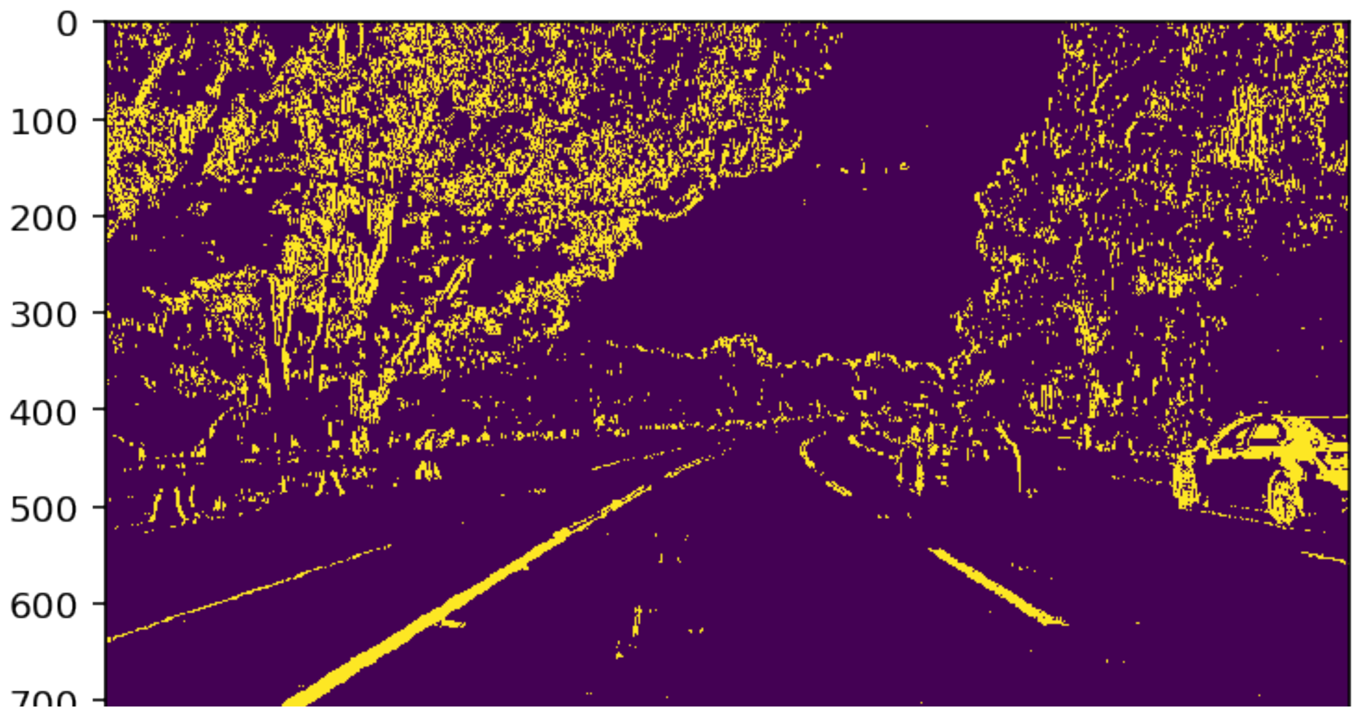


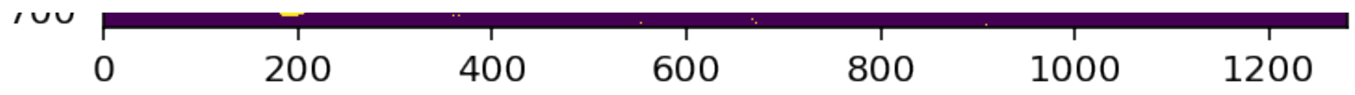

600
700
0
200
400
600
800
1000
1200

L & S:


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Combined:


0
100
200
300
400
500
600
700



In `pipeline.ipynb` you find the applied transforms in lines 5 to 11 of code cell 3, whereas the last line (11) combines the three thresholds (x/y, ls, r) into one.

The functions used for the actual transformation can be found in `threshold.py`. For the x/y gradients the function `abs_sobel_thresh` is used (code lines 4 to 20) which

- converts an input image to grayscale,
- applies the OpenCV function `Sobel` (code lines 9 and 11) in either x or y direction
- and thresholds the output (line 18).

For the ls-thresholding the function `ls_thresh` is used (code lines 69 to 75) which

- converts to HLS (line 70)
- extracts the two color channels (line 71)
- and thresholds the two (line 74).

### 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 `warp_image` in the source file `transform.py`.

The function takes as input an image. I chose the hardcode the source and destination points in the following manner:

```
top_left_dst = [350,0]
top_right_dst = [1000,0]
bottom_left_dst = [350,700]
bottom_right_dst = [1000,700]

top_left = [585,456]
top_right = [700.5,456]
bottom_left = [253,693]
bottom_right = [1071,693]
```

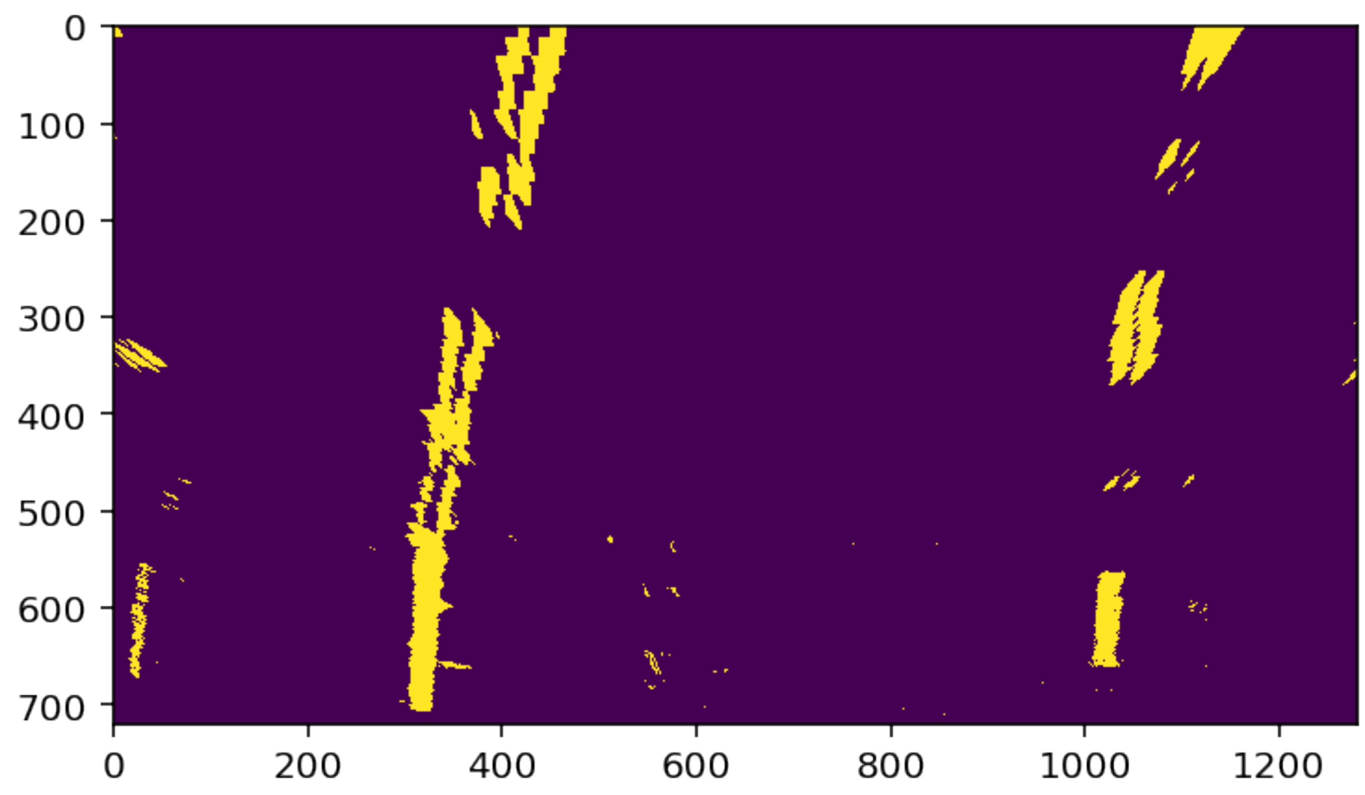
(code lines 6-14)

I chose the source and destination points by using GIMP to exactly measure out the coordinate points.

Given the source and destionation points, I used OpenCV's function `getPerspectiveTransform` to generate the transformation matrices `M` and `Minv` and warped the input image.

I verified that my perspective transform was working by manually inspecting the transformed images and ensuring that the warped lane lines appeared parallel in the output images.

An example input for the initial image above is the following (after undistorting and thresholding):



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

In order to identify lane-line pixels I used the function `detect_lane` in the file `lane_detection.py` . This function uses an histogram of the thresholded (and warped) input to inform the detection of pixels. For this purpose, first the histogram is taken on the lower half of the image (code line 8)

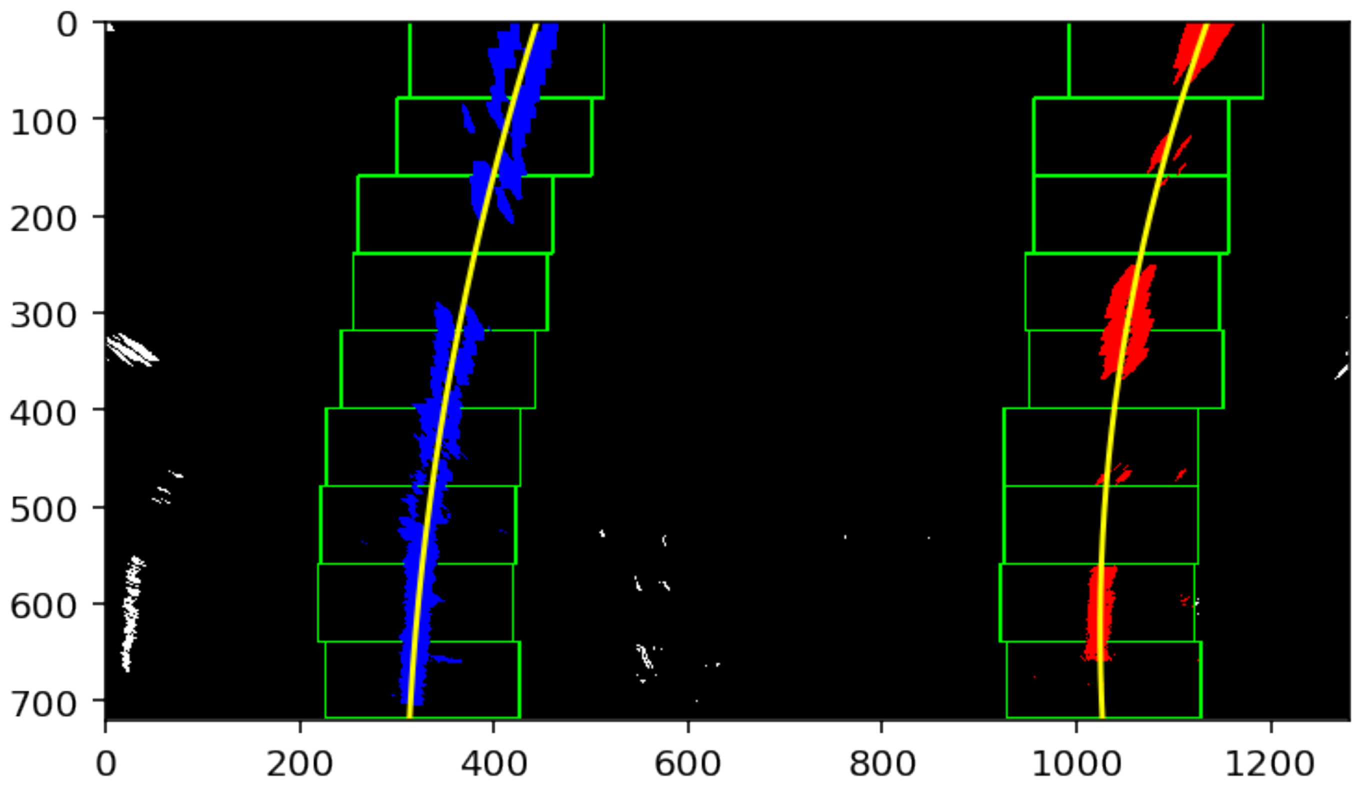
```
histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=
0)
```

Then, the maximum values left and right from center are determined (lines 15 and 16).

```
leftx_base = np.argmax(histogram[:midpoint])
rightx_base = np.argmax(histogram[midpoint:]) + midpoint
```

Afterwards an incremental search is performed over the image. For this purpose, search windows are slided from bottom to top (that's the loop staring in code line 38) and as soon as a threshold value of 50 pixels has been reached (if condition in lines 56 and 58), the search gets recentered to the mean of the found pixels for the left and right lane. Found pixels in a window are remembered (more precisely the index positions of the non-zero pixels are stored; line 53-54).

After all relevant pixels have been determined, the x and y values are extracted and a second-order polynomial is fit to these values (code lines 65-73). The found pixels, the search windows as well as the fit polyonomials are depicted below.



## 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.**

The calculation of the radius of curvature is done in the function `calculate_curvature` in the file `lane_detection.py` (code lines 98-111). First a transformation factor from pixels into meters is defined (lines 101 and 102)

```
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/700 # meters per pixel in x dimension
```

Afterwards, the old x and y values (from the fitted polynomial) are transformed into real world space and a new polynomial is fitted onto these values (code line 105 and 106).

```
left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)
right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*xm_per_pix, 2)
```

Finally, the radius of curvature can be determined in meters using the following formula:

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

Which translates into the following code (lines 108-109):

```
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_f
it_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])
```

The localisation of the car with the respect to center is done in the function `calculate_dist_from_center` in the same file. First, also the conversion factor is determined:

```
xm_per_pix = 3.7/700
```

Afterwards, the lane center is determined (using the x values of the lane at the bottom of the image). Under the assumption that the camera is positioned in the middle of the image, the view_center is simply half the image size.

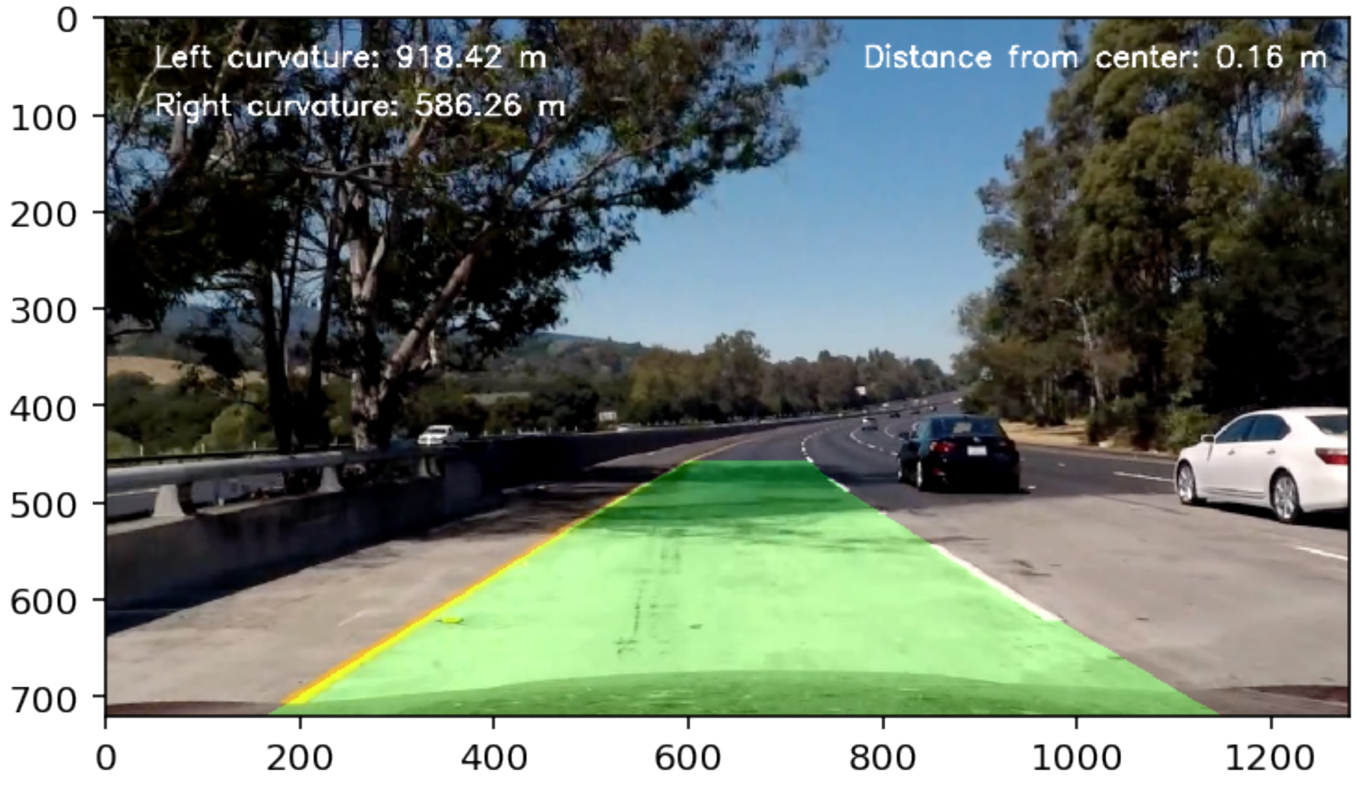
```
lane_center = (left_lane_x + right_lane_x) / 2
view_center = x_len / 2
```

The absolute deviation from the lane center (in real word measures) becomes than simply:

```
np.abs(view_center - lane_center) * xm_per_pix
```

#### 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 114 through 143 in my code in `lane_detection.py` in the function `draw_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]

---

## 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?

Due to time constraints I limited myself to a vanilla implementation of the lane detection that searches each new frame from bottom to top. A possible improvement for that would be to keep track of the previous state throughout the detection process, so that it would be possible to continue searching from where the detection ended in the previous run which would increase performance. Also, I'm still having some problems with detections with varied lightning and color differences of the road. First of all, this could be improved upon by digging even deeper into the color and gradient thresholding (and combination) techniques. In addition, it will make sense to perform smoothing over several images and to discard images with unreasonable parameter values for the fitted polynomials (e.g. when exceeding a certain threshold when comparing coefficient values between two or more runs). For a simpler comparison and access to historic values it would then also make sense to use a class, such as suggested in the lectures, e.g. to abstract away the logic into