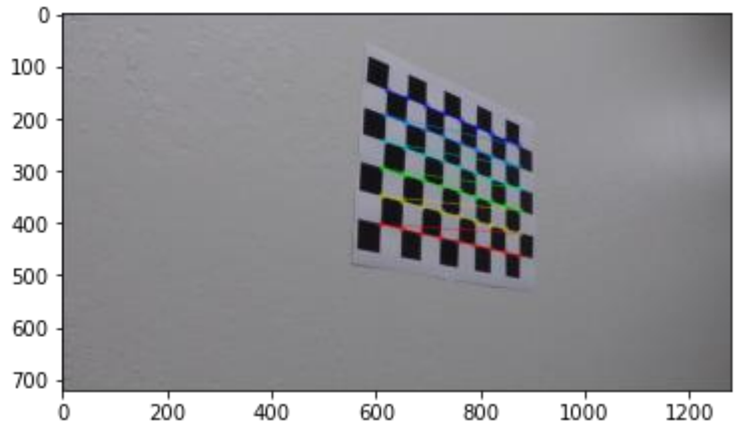
Overall pipeline

1. Camera calibration

Outputs: camera matrix ***mtx*** and distortion coefficients ***dist****.*
Calibration includes:

a. Step through list of calibration images and search for chessboard corners. If found, add object points to ***objpoints*** and corners to ***imgpoints*** (both parameters needed later in cv.calibrateCamera())

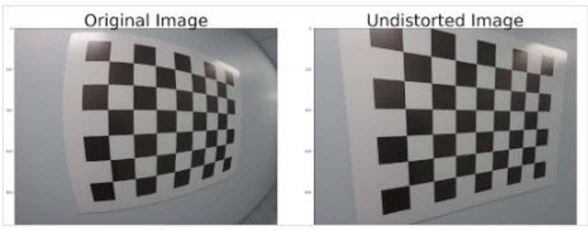


b. Perform camera calibration with the following code:
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img.shape[1:], None, None)

Testing, if camera parameters were successfully obtained:

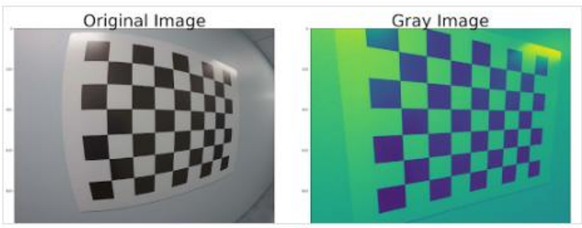
i. Undistort using mtx and dist

undist = cv2.undistort(img, mtx, dist, None)



ii. Convert to grayscale (Why? To find corners easily in the next step)

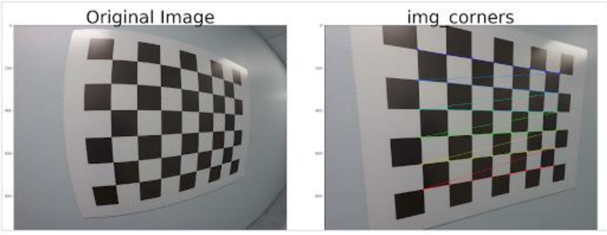
gray = cv2.cvtColor(undist, cv2.COLOR_BGR2GRAY)



iii. Find the chessboard corners and if found, draw them (just for demonstration/fun!)

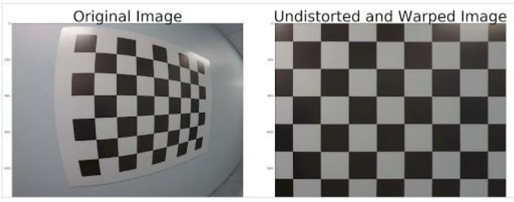
ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)

```
if ret == True:
    img_corners = np.copy(undist) # weil cv2.drawChessboardCorners() das Input-Bild überschreibt
    cv2.drawChessboardCorners(img_corners, (nx, ny), corners, ret)
```



iv. Warp the undistorted image (perspective transform)

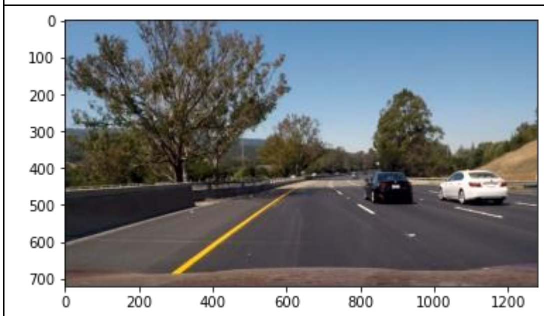
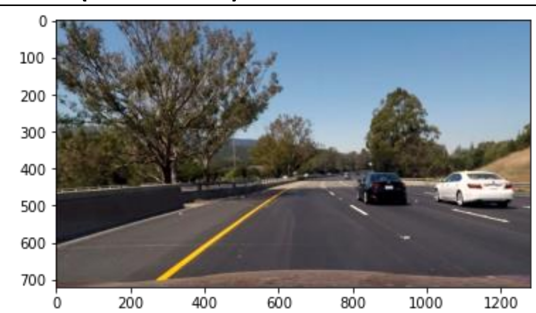
```
offset = 100
img_size = (gray.shape[1], gray.shape[0])
src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-nx]])
dst = np.float32([[offset, offset], [img_size[0]-offset, offset], [img_size[0]-offset, img_size[1]-
offset], [offset, img_size[1]-offset]])
M = cv2.getPerspectiveTransform(src, dst)
warped = cv2.warpPerspective(undist, M, img_size)
```



2. Distortion correction

(using *mtx* and *dist*)

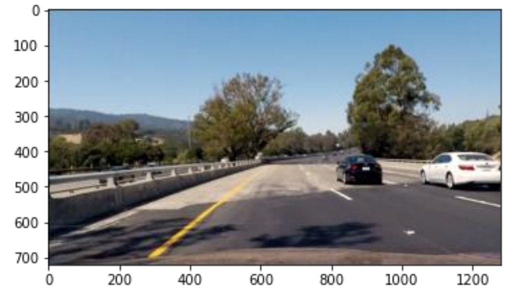
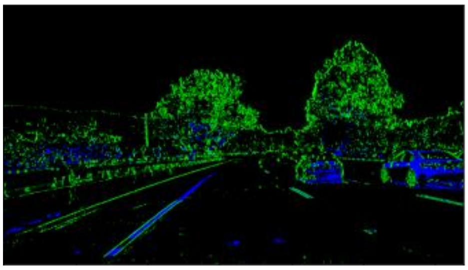
Prerequisite: Camera calibration was done before with cv2.calibrateCamera() (see point 1) giving the camera matrix `mtx` and the distortion coefficients `dist`

Before	**After (undistorted):**
	

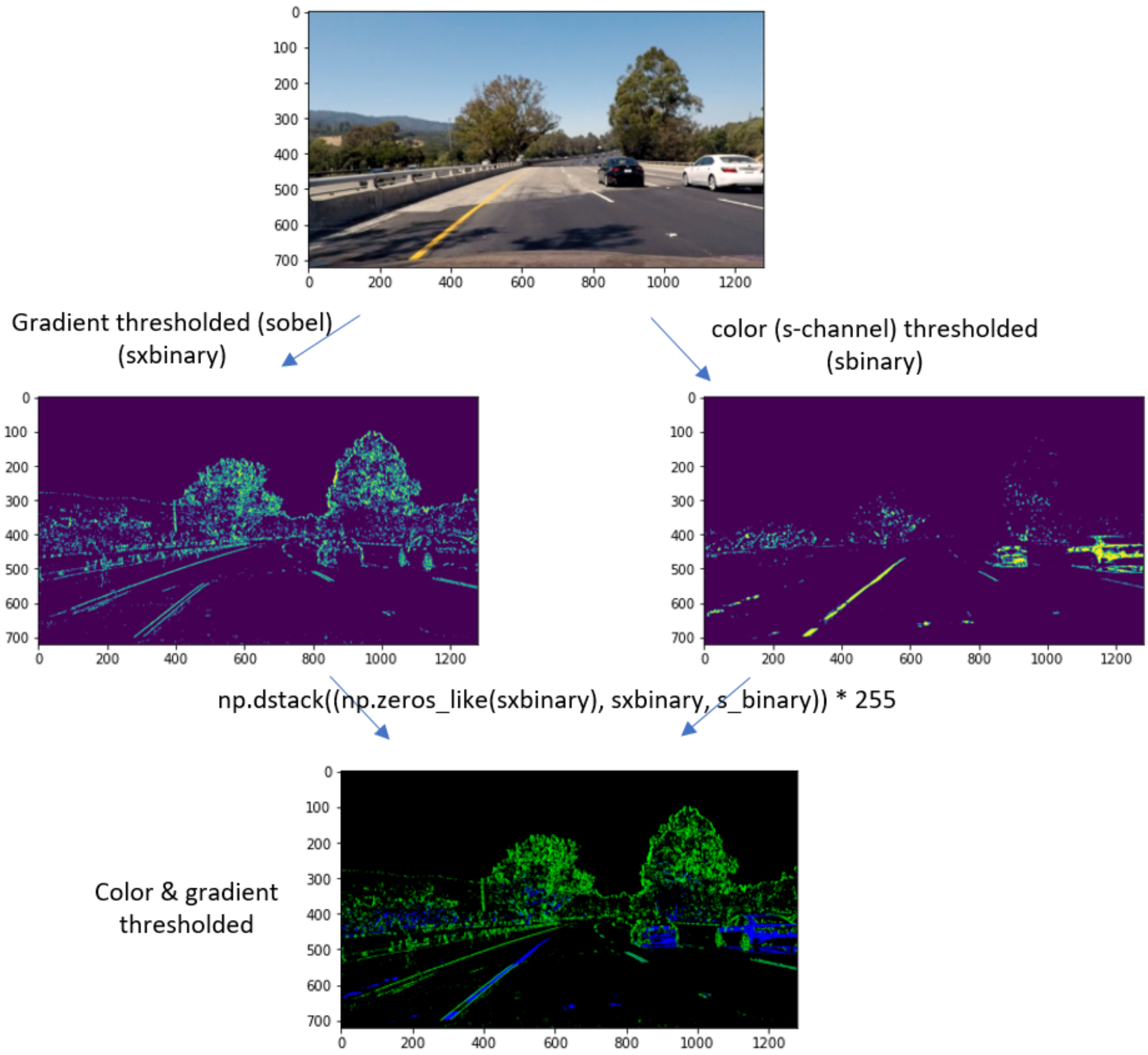
3. Thresholding

Output: binary image (I tried out various combinations of color and gradient thresholds)

(Note: This step is visualized with “test4.jpg” instead of “test6.jpg” as for all the other steps because here you can see the advantages of s-channel when road has bad sun/shadow conditions)

Before	**After:**
	

Steps in detail:



Excursion into sobel operator:

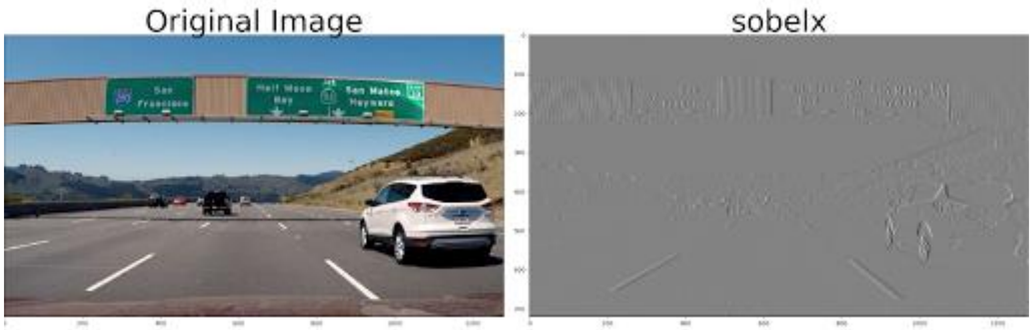
The Sobel operator is a simple edge detection filter that is often used in image processing.

a) Convert to grayscale (Why? Single color needed to calculate derivatives)

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

b) Calculate the derivative in x or y given orient = 'x' or 'y'

sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0)



c) Take the absolute value

abs_sobel = np.absolute(sobelx)

d) Scale to 8-bit (0 - 255) then convert to type = np.uint8

*scaled_sobel = np.uint8(255*abs_sobel/np.max(abs_sobel))*

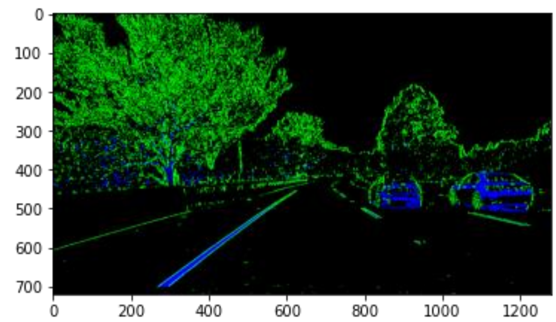
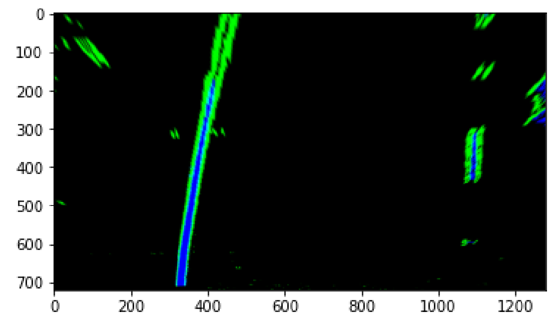
e) Create a binary output image showing where thresholds were met >> create a mask of 1's where the scaled sobel is > thresh_min and < thresh_max

binary_output = np.zeros_like(scaled_sobel)

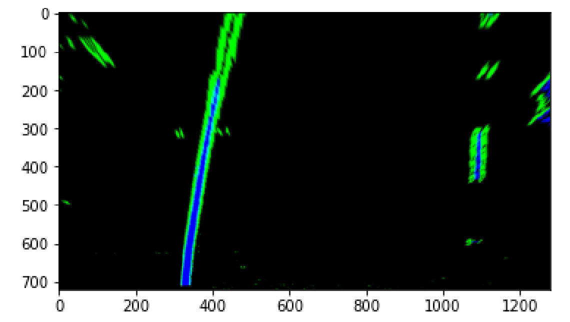
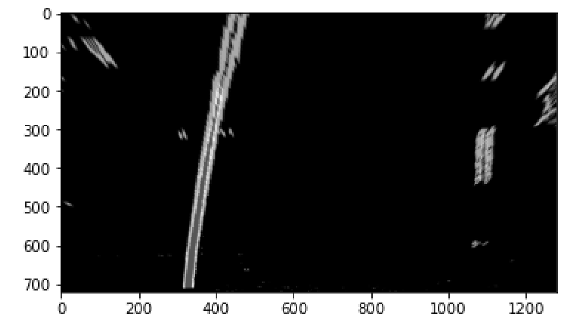
binary_output[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

4. Perspective Transform

First, identifying four source points src (pick four points in a trapezoidal shape (similar to region masking) and after four destination points dst.

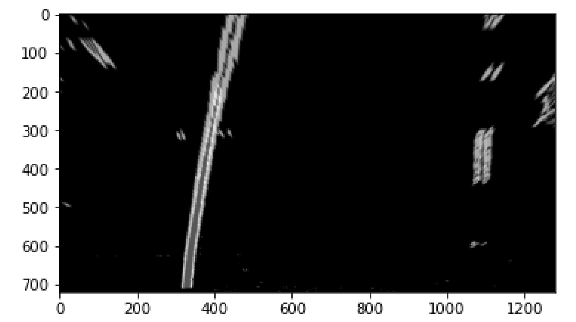
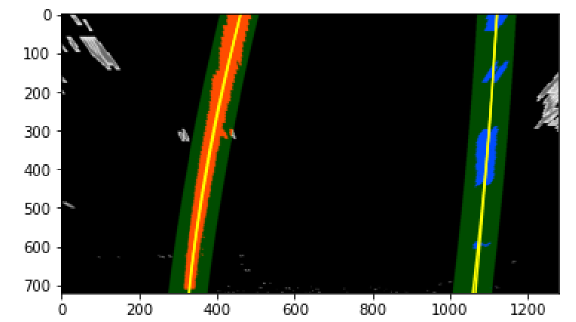
Before	**After:**
	

5. Grayscale

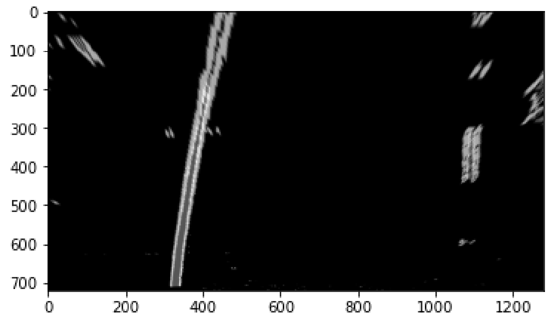
Before	After:
	

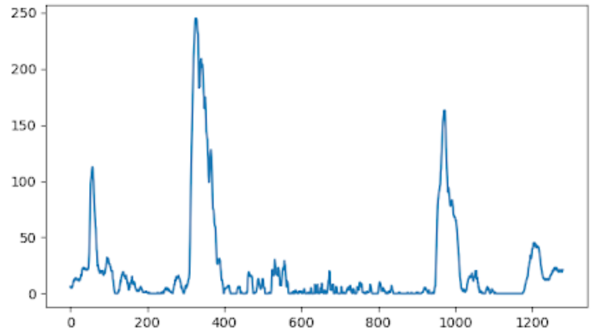
6. Detect lane pixels (sliding windows method) and fit a polynomial to find the lane boundary

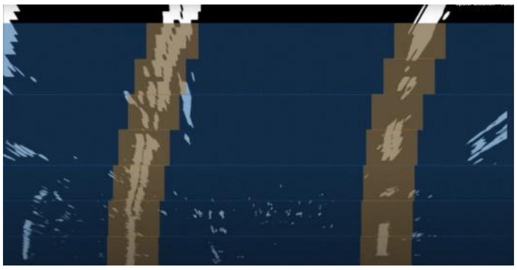
Explicit Decision which pixels are part of the lines and which belong to the left respectively to the right line.

Before	After:
	

Steps in detail:



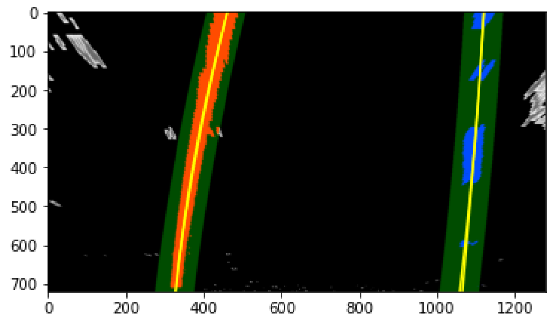
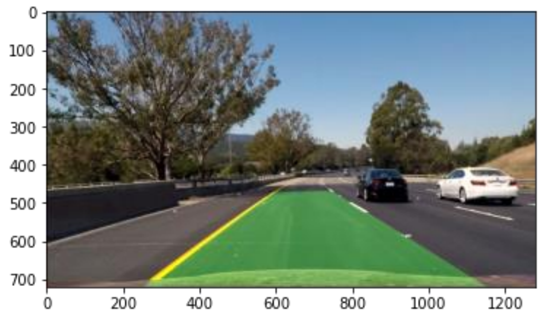




7. Drawing

This includes:

a) Draw the lines on a blank (zeroed) image
b) Warp this image back to original image space using inverse perspective Matrix (Minv)
c) Combine the result with the original image

Before	**After:**
	

Issues Discussion

Description of problems

1) **Hardcoded source points**: In the function warp () the source points are fixed values:
   src = np.float32([[100., 719.],
   [542., 470.],
   [738., 470.],
   [1180., 719.]])

   Each frame uses these points although frames and their respective edges differ. The pipeline still works good especially in short-range lane detection. It could be more robust in long-range detection.
2) **Line detections jump around from frame to frame a bit.**

Solutions to the problems

If I would further develop the project I would take the following steps:

1) **Hardcoded source points**: Instead of manually choose four source points, a smart way to do this would be to use four well-chosen corners that were automatically detected
2) **Line detections jump around from frame to frame a bit:** Smooth over the last n frames of video to obtain a cleaner result. Each time you get a new high-confidence measurement, you can append it to the list of recent measurements and then take an average over n past measurements to obtain the lane position you want to draw onto the image.