Project 4 Writeup - Martin Hintz

Advanced Lane Finding Project - the goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- 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.

Project Code and its Usage

The code is available on Github: https://github.com/marthtz/CarND-Advanced-Lane-Lines-P4
The project was cloned from the official P4 repository. I added the following files:

- In main folder (Python scripts, calibration data and this writeup)
  - cameraCal.py (script for camera calibration)
  - processPipeline.py (script to process images and videos)0
  - cam_cal_pickle.p (saved transformation parameters for camera correction)
  - Writeup.pdf
- In output_images folder (processed test images and videos)
  - project_video_lane.mp4
  - challenge_video_lane.mp4
  - harder_challenge_video_lane.mp4
  - project_video_lane_debug.mp4 (plus debug info screen)
  - straight_lines1_lane.jpg & straight_lines2_lane.jpg
  - test1_lane.jpg to test6_lane.jpg
- In output_images/writeup_images folder(images mentioned in this writeup)
  - ColorSelectionTool.jpg
  - calibration1_undist.jpg
  - straight_lines1_undistorted.jpg
  - test6_original_thresholded.jpg
  - straight_lines1_original_lines.jpg
  - straight_lines1_warped_lines.jpg
  - straight_lines1_binary_warped.jpg
  - test2_left_binary_line.jpg
  - test2_binary_warped.jpg
  - video_right_binary_line.jpg
  - test2_lane.jpg
  - video_debug.jpg

Usage of the Python scripts:

- cameraCal.py
  This scripts runs the camera calibration on the provided calibration images. Execution is done by just calling .py. All calibration images must be present in the subfolder ‘camera_cal’.
  The script saves the parameters used for undistorting images as ‘cam_cal_pickle.p’ in the main folder. An example of an undistorted calibration image is saved to “output_images”
- processPipeline.py
  This script runs full lane line detection on the provided images or videos. In addition, debug information can be created. Executing is done by calling the script with input images/videos as parameter. In addition, if the first argument is ‘debug’, additional output is generated.
  Examples:
  - `python processPipeline test_images/image1.jpg test_images/image2.jpg`
  - `python processPipeline debug project_video.mp4 challenge_video.mp4`

Rubric Points

1. **Camera Calibration**

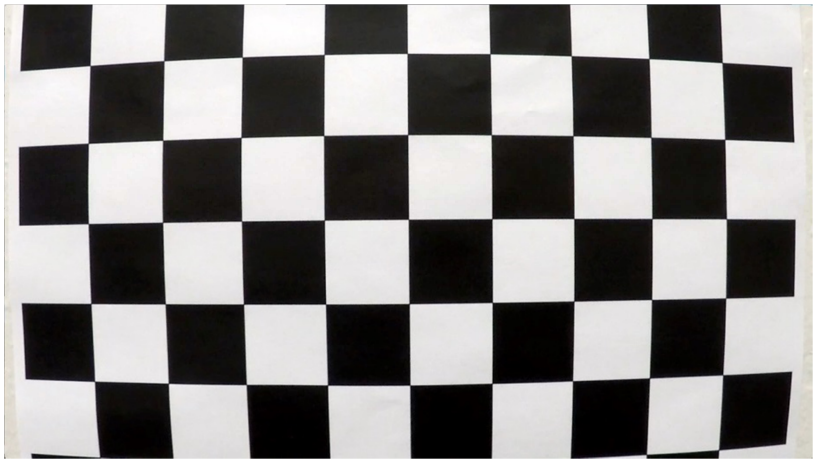
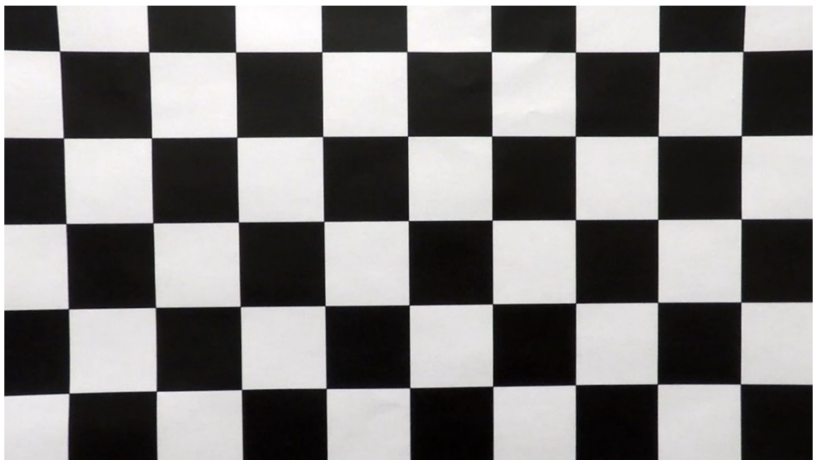
   This code can be found in 'cameraCal.py'.

   I started by looking at the calibration images and counted the number of inner corners of the chessboard. For this project the chessboard has 9 corners on the x axis and 6 on the y axis.

   Next step was to prepare "object points", which will be the (x, y, z) coordinates of the chessboard corners in the real world, however, it's assumed that the chessboard is flat in each image. Hence the z coordinate will be set to 0. 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. imgpoints will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

   I then used the 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.

   Example:

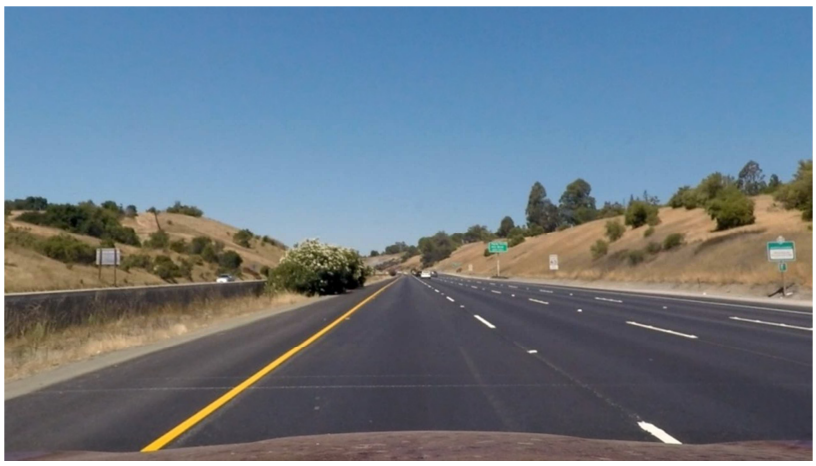
Original Image calibration1.jpg	
Undistorted Image calibration1_undist.jpg	

2. **Pipeline (single images)**

2.1. Distortion correction on test images

This code can be found in 'processPipeline.py', function 'undistortImg()', line 383.
The saved mtx and dist values from the previous done camera calibration can be reused to undistort the sample images provided in "test_images" since they're taken by the same camera. Similar
The distortion can be especially being seen at the bottom corners and around the engine hood especially above the hood where there's a mirrored image of the dashboard. In the undistorted image the "horizontal" line is quite curved while in the undistorted picture the line is more or less horizontal.

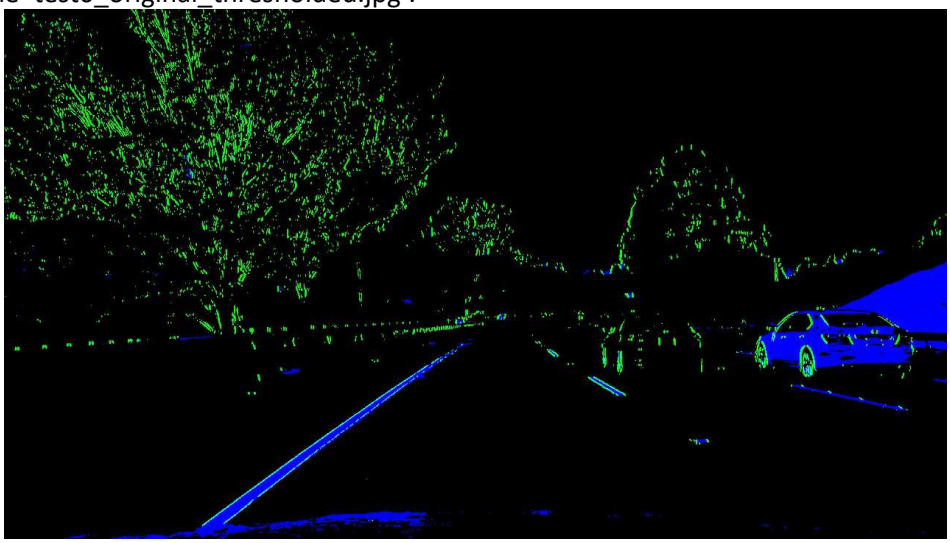
Example:

Original Image straight_lines1.jpg	
Undistorted Image straight_lines1_undistorted.jpg	

2.2. Generate a thresholded binary image

This code can be found in 'processPipeline.py', function 'selectHSVSobelX ()', line 132.
To generate a thresholded binary image I combined the result of a color selection (for white and yellow, corresponding to the lane line colors) and a gradient detection. See following subsections for further details on individual items. I used a simple or operation to combine the color mask and the gradient mask and made sure the mask contained only 1s and 0s. For debug purposes I also generated an RGB image that shows the color mask in blue and the gradient mask in green.

Example 'test6_original_thresholded.jpg':



2.2.1. Color selection

I chose to do a color selection for yellowish and whitish colors, similar to the colors to be expected for the lane lines. For this purpose I used the HSV color space as it allows you to select a large range of theses colors in different conditions (e.g. when there are shadows etc.).
I used the OpenCV function "inRange" and specified a lower and an upper threshold to select only the wanted colors. I then used a bitwise or operation to combine both color masks.

	Lower HSV threshold	Upper HSV threshold
Yellow	0, 70, 130	25, 255, 255
White	0, 0, 175	40, 25, 255

2.2.2. Gradient detection

The gradient selection mask is based on Sobel derivatives in x direction only. I experimented with y direction as well but found out it gives only very little extra info. This is kind of expected as we're looking for lane lines, which are more or less constant in y direction as they're following the same x value along the y axis.
I thresholded the found gradients by their magnitude and used only the ones which were in the range of 50 to 225.

2.3. Perspective transform

This code can be found in 'processPipeline.py', function 'warpImage()', line 62.
I defined my points in clockwise direction starting at the top left and ending at bottom left. Although the image size is 1280 x 720 (x, y), the valid pixel indices range from 0-1279 on the x axis and 0 – 719 on the y axis.
Beside the warped image the function also returns the inverse transformation matrix to be able to draw the detected lane onto the camera image in a later stage.
I use hardcoded source points by manually selecting the points from a test image with straight lines. I chose the destination points by using min/max of the original image for the y axis and selected the x values to be in the center of each half of the image.

Destination points calculation:

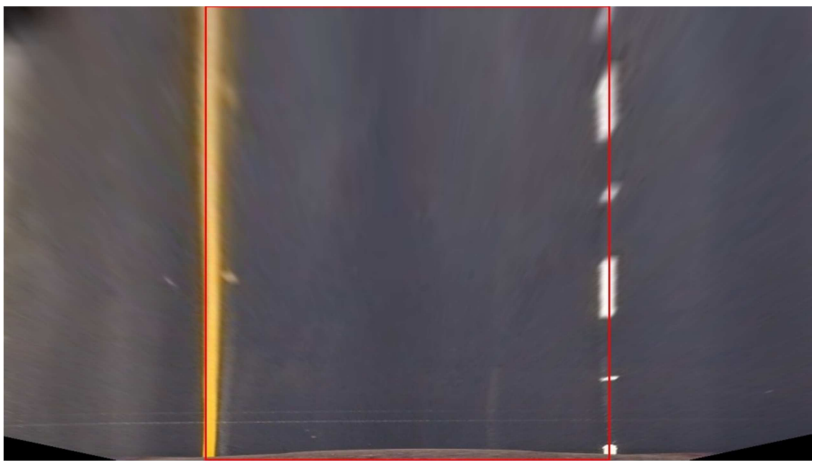
```
midPoint = img_size[1]//2
offset = midPoint//2
tl = [offset, 0]
tr = [img.shape[1]-1-offset, 0]
br = [img.shape[1]-1-offset, img.shape[0]-1]
bl = [offset, img.shape[0]-1]
dst = np.float32([tl, tr, br, bl])
```

This resulted in the following source and destination points:

Position	Source	Destination
Top left	588, 455	320, 0
Top right	694, 455	959, 0
Bottom right	1100, 719	959, 719
Bottom left	200, 719	320, 719

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

Example:

Original undistorted Image straight_lines1_original_lines.jpg	
Warped undistorted Image straight_lines1_warped_lines.jpg	

Warped undistorted binary Image straight_lines1_binary_warped.jpg	

2.4. Lane-line identification and polynomial fit

This code can be found in 'processPipeline.py', function 'findLaneLine()', line 193.

Lane line detection is split into two variants. First variant is used whenever no lane lines have previously been detected. This variant uses a sliding window to look for lines in the whole image. Variant 2 is used when lane lines were detected in the previous image/frame and just searches lines within a specified margin from the previous detection.

Both variants are used to identify pixels in the image that are used to fit a 2nd order polynomial with independent variable 'x' and dependent variable 'y'. This is done by the numpy function 'polyfit' and returns the line we detected as the lane line.

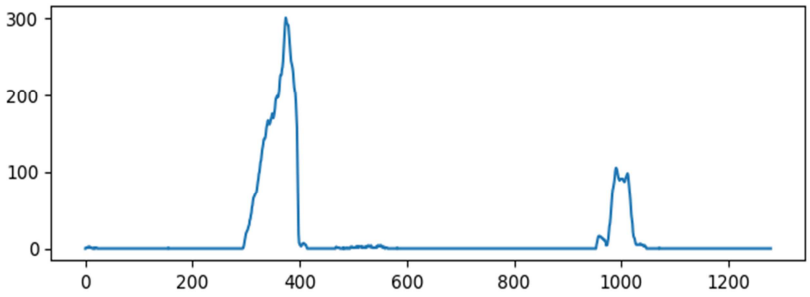
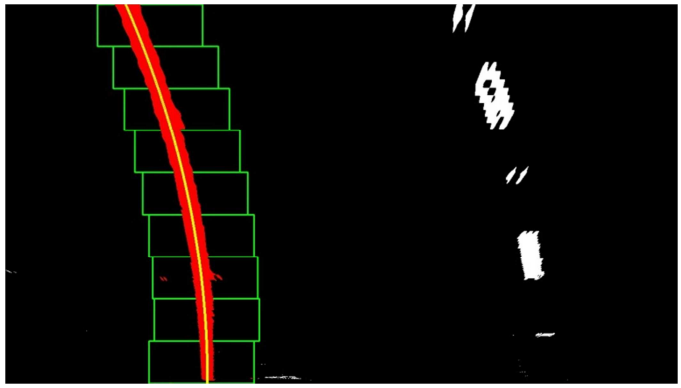
2.4.1. Sliding Window Search

When no previous line has been found or if it's the very first image/frame it's unknown where the lane lines are in the image. Hence, the whole image is to be searched. The search is done by sliding a window over the image with a defined overlap and accumulating the pixel values. Each window spans the whole x axis but only a defined height on the y axis. For each window a histogram of active pixels is calculated. The peaks in this histogram are then used to set the center for a region, which defines the pixels that are being used to fit the lane line.

The 1st image in the table below shows the binary image that is to be searched. The 2nd image shows an example histogram of a single window. The 3rd image shows all regions for the left lane line (green). The center points of these regions have been calculated as the peak of the histogram for the left side. The dimensions for a region are predefined. All active pixels (red) within these regions are used to fit the lane line (yellow).

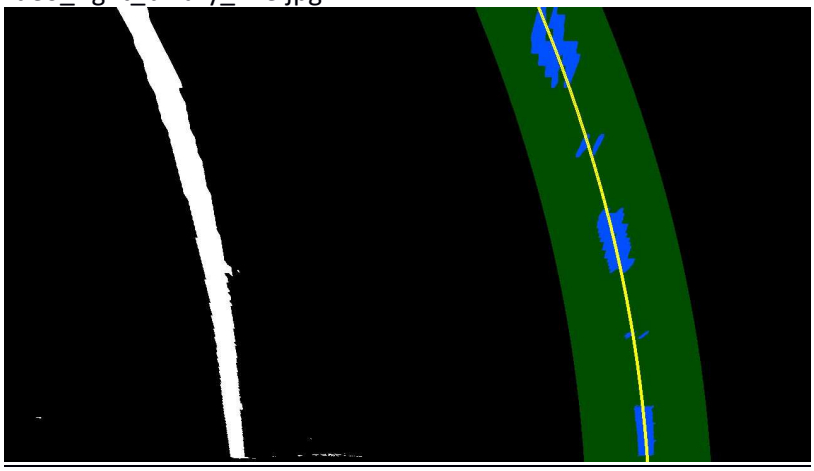
This method assumes that the previous processing steps that created the binary image worked well enough to remove most information from the image and just keep lane line relevant information.

Example for first window sweep at the bottom:

test2_binary_warped.jpg	
Corresponding histogram of a single window	
Regions (green) with active pixels (red) used for fitting the lane line (yellow) test2_left_binary_line.jpg	

2.4.2. Pixel detection with previously detected lane line

In case a lane line was found in an earlier detection it's much simpler and computational faster to select lane line relevant pixels. The method just uses the quadratic function of the previous detected line and adds all pixels within a specified margin of this line to the new fitting function. Example 'video_right_binary_line.jpg':



2.5. Radius of lane curvature

This code can be found in 'processPipeline.py', function 'getCurve()', line 328.
As 2nd order polynomials are used to describe the lane lines with f(y) = A*y^2 + B*y + C the radius of curvature can be described as (from Udacity lesson):

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

The final step is to define where in the fitted line to calculate the curvature. I chose to use the bottom of the image (where y is maximum).

As these calculations are done in pixel space they have to be converted to real space using a factor to convert pixel in meter. For the x axis I measured the number of pixels between the lanes (640 pixel for the set lane width of 3.7m) and for the y axis I modified the value Udacity provided (to 720 pixel for 40m distance) as my upper transformation points are farther in the distance than the Udacity ones.

2.6. Position of the vehicle with respect to center

This code can be found in 'processPipeline.py', function 'getOffset()', line 360.
The camera is mounted in the center of the vehicle. Therefore, the offset of the vehicle with respect to the lane center is simply calculated by taken the difference between the image center and the lane center. As this is again in pixel space it has to be converted to real world space using the same x axis factor as for the radius of lane curvature.

2.7. Full processing pipeline

This code can be found in 'processPipeline.py', function 'pipeline()', line 482.

2.7.1. Preprocessing

This code can be found in 'processPipeline.py', function 'pipeline()', line 492 to 518.
My full processing pipeline uses all the previously described actions to detect lane lines (and their features). First, the camera is corrected by distorting the input image. Next step is to create a binary image by using color and gradient masks. Then, a birds-eye view of the lane is created. This also removes any unwanted parts of the image (like the sky) and focusses on the lane. With this information the actual lane line detection can start.

2.7.2. Lane detection and sanity check

This code can be found in 'processPipeline.py', function 'pipeline()', line 522 to 637.
I chose to do left and right lane detection separately as at one stage I used a sanity check that compared the detected lane line curvature with previous fits. However, that proved to be not mature enough so I removed it for the time being.
After the lane line detection I added a sanity check to check if the detected lines are kind of parallel. I decided that line are parallel if the smaller curvature radius is within a 50% margin of the larger curvature radius. In addition, if the curvature radius is very large (and basically not a curve) and the signs of left and right lines are different, I defined these lines as parallel too.

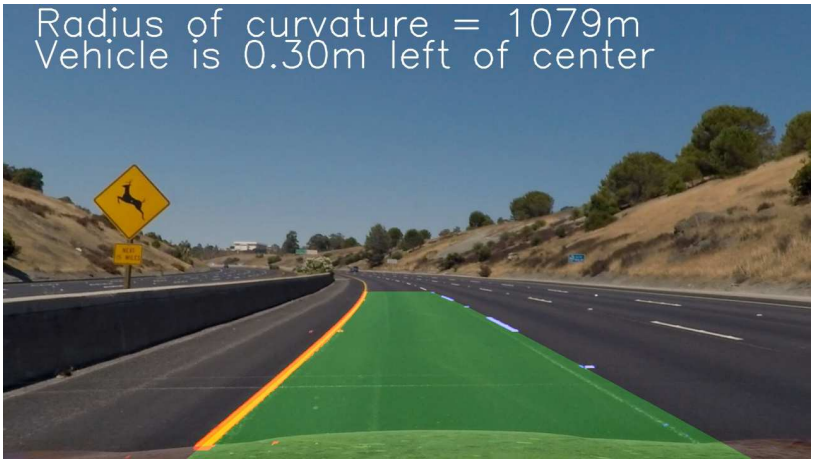
2.7.3. Lane line update and vehicle offset from center

This code can be found in 'processPipeline.py', function 'pipeline()', line 662 to 678.
In case the sanity check on the detected lines was successful, the newly detected line fit and curvature are added to the previous detections with a factor of 0.4 while the curvature uses a factor of 0.2. This is done to make the lines and the curvature values smoother and more stable. While the images can vary quite a bit, e.g. due to bumps on the street, the real world lane isn't.
A new fitted line is then generated from the updated and weighted polynomial fit.

2.7.4. Add lane detection to camera image

This code can be found in ‘processPipeline.py’, function ‘pipeline()’, line 682 to 705 and in function ‘drawLane()’ line 413 and function ‘addText’()’ line 453.
Final step in the processing pipeline is to add the detected lane by overlaying it with the original camera image. In addition, text containing lane (radius of curvature) and vehicle features (lane center offset) is added to the camera image.

Example ’test2_lane.jpg’:



3. **Pipeline (video)**

The pipeline used for video processing is similar to the previously described pipeline on single images. It only differs in some debug options.

4. **Debug tools**

This project has quite a lot of processing steps. Therefore, I added various debug options at every stage to check the performance of each stage.

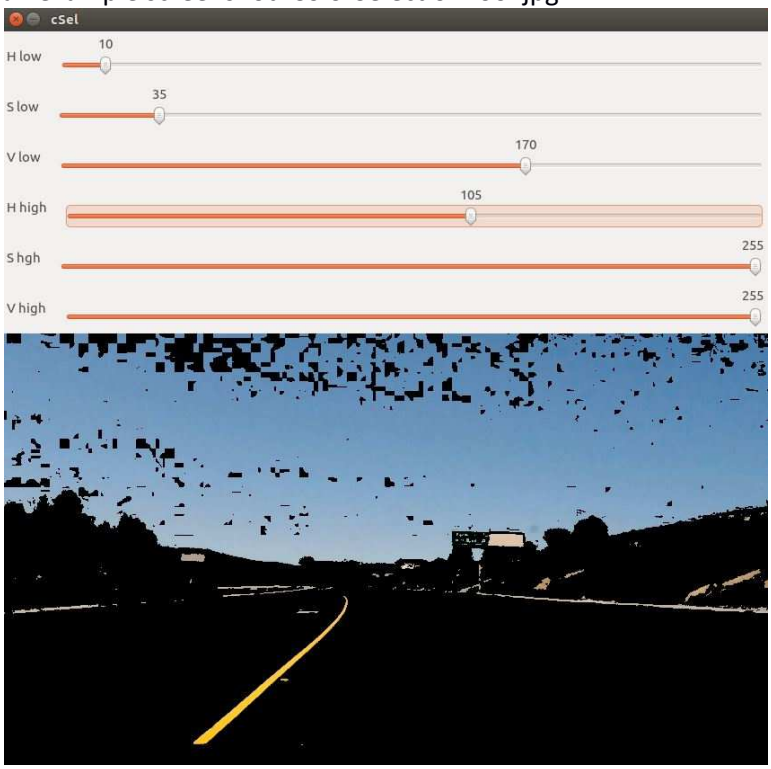
- For single images I simply saved an image after each processing step.
- To be able do debug the videos I created a debug video that combines all images of intermediate processing steps plus some textual output (like detection successful, left/right curvature radius). This allowed me to understand the decision making process for every single frame in the videos. This code can be found in ‘processPipeline.py’, function ‘debugImg()’, line 747

Example ‘video_debug.jpg’:

4.1. **Interactive color and gradient threshold**

A very important part of the preprocessing is the creation of the binary mask by using color selection on the images. It can be quite time consuming to find good thresholds. Therefore, I created a little tool in python that allows the user to interactively play with threshold to find the optimal values for an image. The tool was inspired by a similar tool a student made for project 1 where it was used for edge detection. The tool is available on GitHub: https://github.com/marthtz/colorSelect

Here's an example screenshot 'ColorSelectionTool.jpg':



5. **Problems**

The most difficult thing in this project probably was to find good thresholds for the color/gradient selection. The values I finally chose work ok but not great. This is also the reason why the lane lines in the challenge video under the bridge are not detected very well and the harder challenge video is really bad. I was able to modify the thresholds so that the challenge video is performing very good, but these thresholds would make the project video fail. In my mind, the thresholds have to be set more tightly and another approach (other color spaces for example) has to be used in addition to create a robust binary image that can be used for lane line detection.

Another hard bit was the sanity check on the detected lane lines. I first started on comparing a detected line with a previous detection. It turned out that this was very hard to use. Not only the radii where changing quite a lot but also sign changes (basically when there was no curve but a straight) gave me a hard time. Finally only checking if left and right lines are parallel was much more robust to get a working solution for this project. However, for a product used in a vehicle much more sanity checking will be required to create a trusted result. Adding outlier rejection would have increased the performance and stability as well in my mind.

I was also surprised of the speed of the lane line detection. My pipeline was achieving 10 to 12 frames per second, which is way too slow for lane line detection in real-time. There are several reasons for this (not optimized code, Python, single CPU core usage), however, still surprising.

Finally, it has to be said that this model is most likely only applicable under the good environmental conditions of the given videos. Different weather conditions as well as different terrain will probably give it a hard time. This can already be seen in the harder challenge video where there are very sharp curves over a short distance. A quadratic approximation is not possible anymore and a higher order polynomial would be required to get an appropriate fit.