Finding Lane Lines with Image Processing

14 Mar 2017 | Tags : python (/blog/tags.html#python) opencv (/blog/tags.html#opencv) image_processing (/blog/tags.html#image-processing) self_driving_cars (/blog/tags.html#self-driving-cars) matplotlib (/blog/tags.html#matplotlib) sobel_operator (/blog/tags.html#sobel-operator) convolution (/blog/tags.html#convolution) perspective_transform (/blog/tags.html#perspective-transform) camera_calibration (/blog/tags.html#camera-calibration)

Finding lane lines in camera images can be more involved than you think. This post will go through a slightly advanced methodology which uses image calibration and transformation, Sobel operator and color thresholding to clean the images, and sliding window search to find and track lane lines.

The process we will fallow can be summarized as:

- 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.

You can find the full code, images and videos in this **REPO (https://github.com/cmlpr/behavioral-cloning)**. The summary of the files and folders int repo is provided in the table below:

File/Folder	Definition
`process.py`	Main python file that runs the program
`flib.py`	Function library - includes all the helper functions and the main pipeline
`LineTracker.py`	Includes the class to store tracking information and the window search function
`corners.p`	The pickle file which contains the corners of the chessboard
`calibration.p`	The pickle file which stores the camera calibration matrix
calibration	Folder with chessboard images that will be used in calibration process
test_images	Folder with road images used to test the pipeline
output_images	Folder to store all output images
test_videos	Folder with the test videos
output_videos	Folder to store output videos

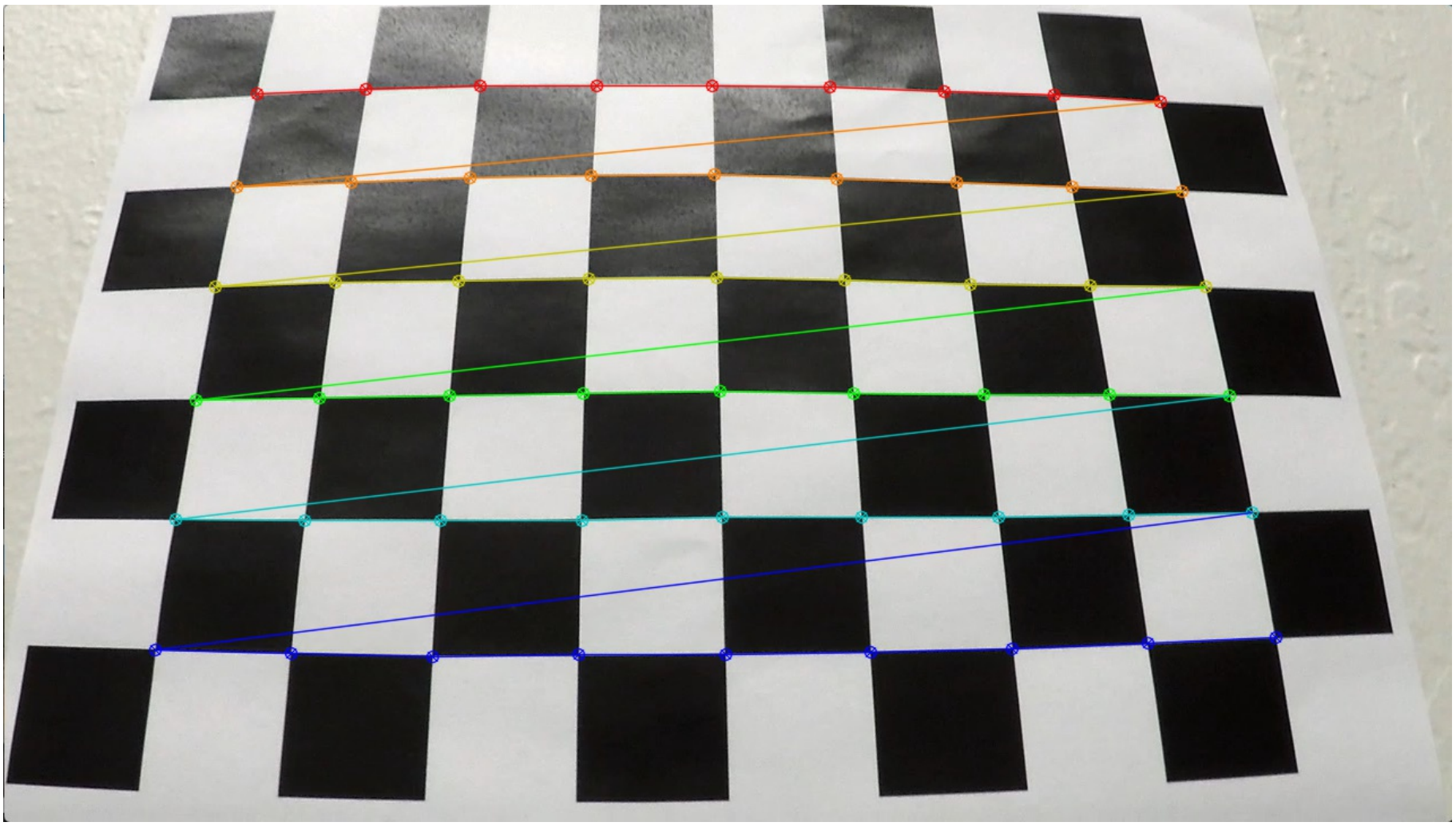
Now let's review the steps one by one.

Camera Calibration

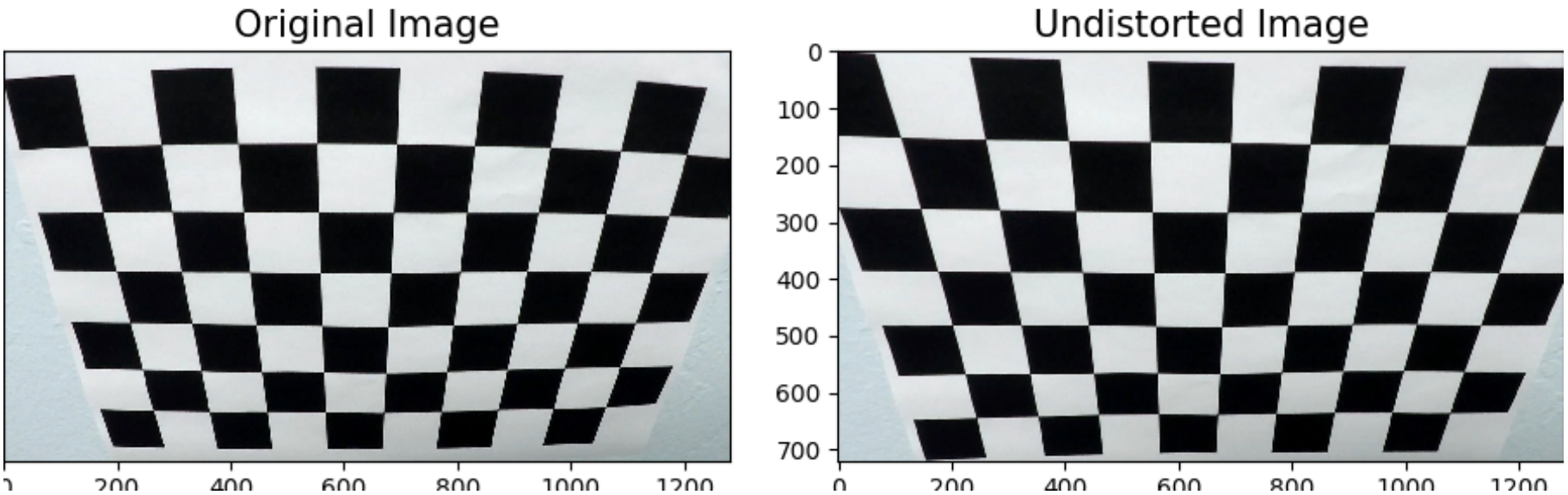
Two functions are used for camera calibration: `get_points` and `get_calibration`. Both of these functions are in the `flib.py` file.

Calibration process uses many images of a chessboard drawing taken from different angles and distances. Using `cv2.findChessboardCorners()` function we find and store the corners of the chessboard in the images in `corners_array`. This function requires you to provide the corner counts in the horizontal and vertical directions which I stored in `corner_count` tuple. We also need an array with the indices/coordinates for each corner in each image. This index information is stored in `index_array`. We append 2D matrix of corner coordinates into this array as many as calibration images. The coordinates are 2D since we assume the chessboard is fixed on the (x, y) plane at z=0.

`get_points` function goes through all the images in the calibration folder and tries to find the chessboard corners. If the search is sucessful, it will add the corners to the `corners_array`, draw these points on the image and saved the modified image into `output_images` folder. Sometimes the process is not sucessful as not all the corners appear in some images due to zoom level. An example calibration images with detected corners:



Get points stores both arrays in `corners.p` pickle file so that we don't have to do this everytime. Now in the following process `get_calibration()` function reads the arrays from the pickle file and calls `cv2.calibrateCamera` function to get the camera matrix, distortion coefficients and position of the camera in real world with rotation and translation vectors. It then stores these output values in `calibration.p` pickle file and uses the distortion information to undistort each calibration image before exiting. Here is an undistorted calibration image:



Also an undistored test image which shows some changes around the edges.

Image Thresholding Functions

The next step is to prepare the functions used in color transforms, gradients, etc., that will help us create a thresholded binary image. These functions are in `flib.py` file and here is a table:

File/Folder	Definition
abs_sobel_thresh	Applies Sobel in x or y with a given kernel size and threshold range for absolute vaue masking
mag_thresh	Apples Sobel in both x & y with a given kernel size and threshold range for magnitude masking
dir_threshold	Applies Sovel in both x & y with a given kernel size and threshold range for directional masking
gray_select	Converts the image to grayscale and applies mask based on the threshold values provided
rgb_select	Thresholds a specific channel in RGB colorscale based on the range provided
hls_select	Converts the image to HLS scale and applies threshold to desired channel with the range provided
hsv_select	Converts the image to HSV scale and applies threshold to desired channel with the range provided

Sobel is a gradient calculation methodoloy. Sobel operators with kernel size of 3 is shown below. The kernel will be translated along the image matrix directions and at each point the element-wise product of the kernel and corresponding image pixels will be calculated. The resulting matrix which is at the same size of the kernel will be

summed to get the derivative (gradient) at that point. When S_x is applied to a section of an image where pixel values are rising from left to right, then the derivative in the x direction will be positive.

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

`OpenCV` has `cv2.Sobel()` function to get the gradients. Here is an example

```
sobel = cv2.Sobel(gray, ddpeth = cv2.CV_64F, dx = 1, dy = 0, ksize=3)
```

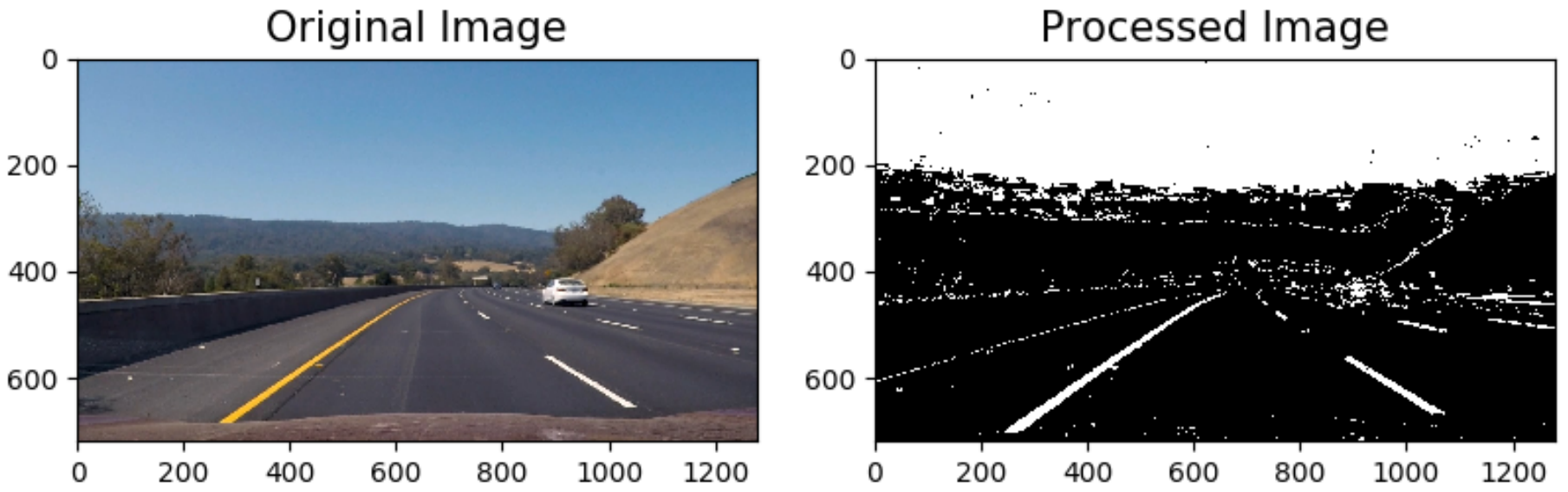
This function requires a grayscale image, desired direction values (dx, dy) and the kernel size. Using the output values and desired threshold range (max, min gradients values), we can get a masked binary image. I have three options in the `flib.py` file. In `abs_sobel_thresh()` function, we can apply thresholds to the absolute value of either x or y gradients. `mag_threshold` calculates the magnitude of the gradients at each point and then masks the image based on the desired range. `dir_threshold` looks at the angle formed by the x and y gradients (arctan2(sobely, sobelx)) and maskes based on the angle thresholds.

Another method for masking images to get clean images is to use colorscales. Converting the image to `gray, rgb, hsv, hsl` are possible options here. Depending on the features we want to keep or eliminate we can use one or combination of these transformations and apply thresholding to different channels.

The methodology I followed is as follows:

```
gradx = abs_sobel_thresh(img, orient='x', sobel_kernel=3, thresh=(12, 255))
grady = abs_sobel_thresh(img, orient='y', sobel_kernel=3, thresh=(25, 255))

sthresh = hls_select(img, 's', thresh=(100, 255))
vthresh = hsv_select(img, 'v', thresh=(50, 255))

combined = np.zeros_like(gradx)
combined[(gradx == 1) & (grady == 1) |
            ((sthresh == 1) & (vthresh == 1))] = 255
```

Thresholds to absolute value of x and y gradients, saturation channel of the HLS color space and value channel of the HSV color space were combined to get the thresholded image example below. This is a process which heavily depends on a trial-error on test images. There are many combinations that one can pick.



Perspective Transform

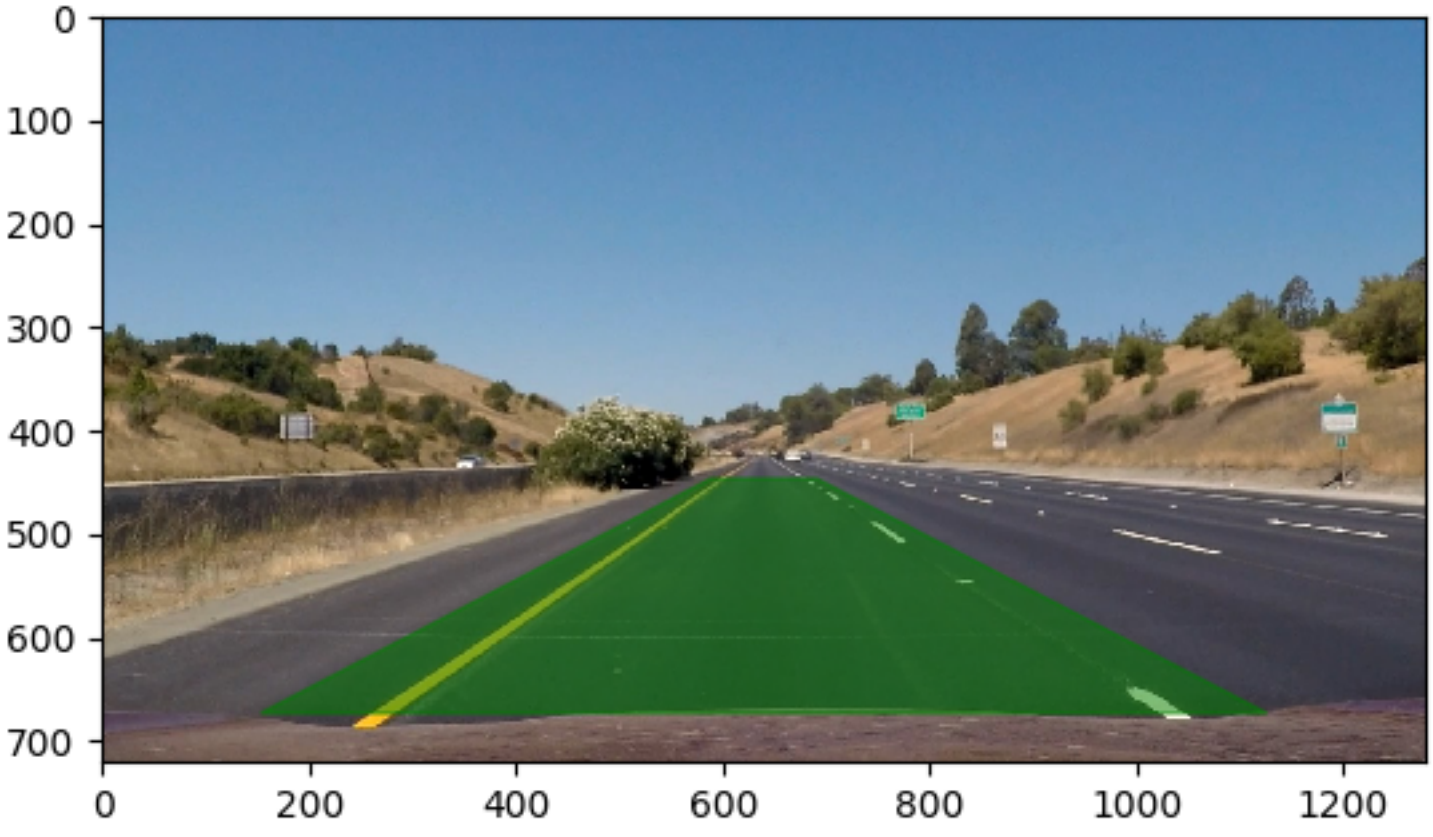
In regular undistorted images lane lines looks to be converging at far disctances from the car. We know that lane lines are almost parallel to each other in general. This is known as the perspective effect. If we transform the image so that we get a birds-eye-view of the road, it would be much easier to detect lane lines knowing that both lines will be parallel to each other. Tranformation will be necessary to know the lane curvature as well.

The code for my perspective transform is included in main `pipeline()` function in `flib.py` file between lines 370 and 417. This section uses the thresholded image (`combined`), as well as source (`src`) and destination (`dst`) points. I chose to hardcode the source and destination points in the following manner:

```
box_width = 0.76 # fraction of the bottom edge length of the trapezoid w.r.t image w
idth
mid_width = 0.08 # fraction of the top edge length of the trapezoid w.r.t image widt
h
height_pct = 0.62 # fraction of the top edge veritcal position of the trapezoid w.r.
t image height measured from top
bottom_trim = 0.935 # fraction of the bottom edge veritcal position of the trapezoid
w.r.t image height measured from top

src = np.float32([[img.shape[1] * (.5 - mid_width / 2), img.shape[0] * height_pct],
                  [img.shape[1] * (.5 + mid_width / 2), img.shape[0] * height_pct]
,
                  [img.shape[1] * (.5 + box_width / 2), img.shape[0] * bottom_trim
],
                  [img.shape[1] * (.5 - box_width / 2), img.shape[0] * bottom_trim
]])

# offset adjusts the shrinkage of the warped image - larger is shrunken more
offset = img.shape[1] * 0.25

dst = np.float32([[offset, 0],
                  [img.shape[1] - offset, 0],
                  [img.shape[1] - offset, img.shape[0]],
                  [offset, img.shape[0]]])
```

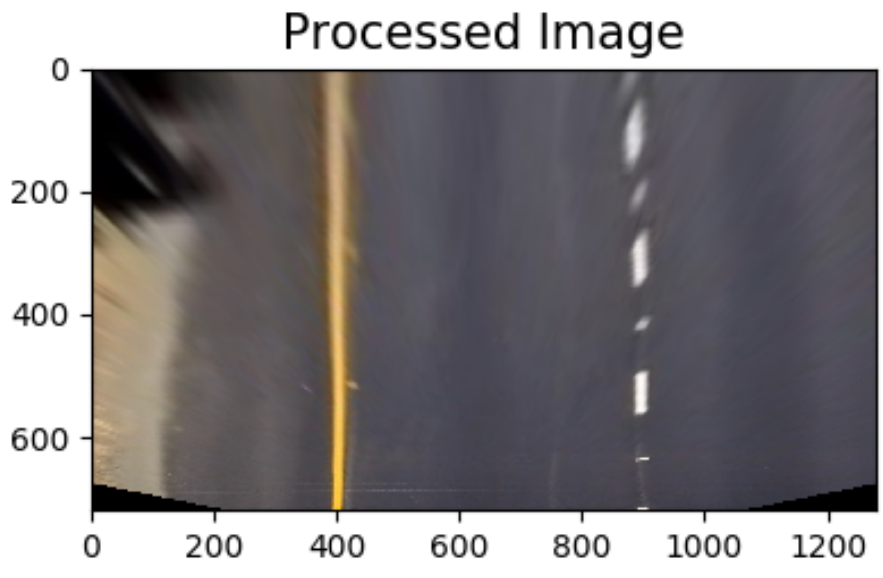
For the source points I defined fractions with respect to the image size. This resulted in the following source and destination points:

Source	Destination
588, 446	320, 0
691, 446	960, 0
1126, 673	960, 720
154, 673	320, 720

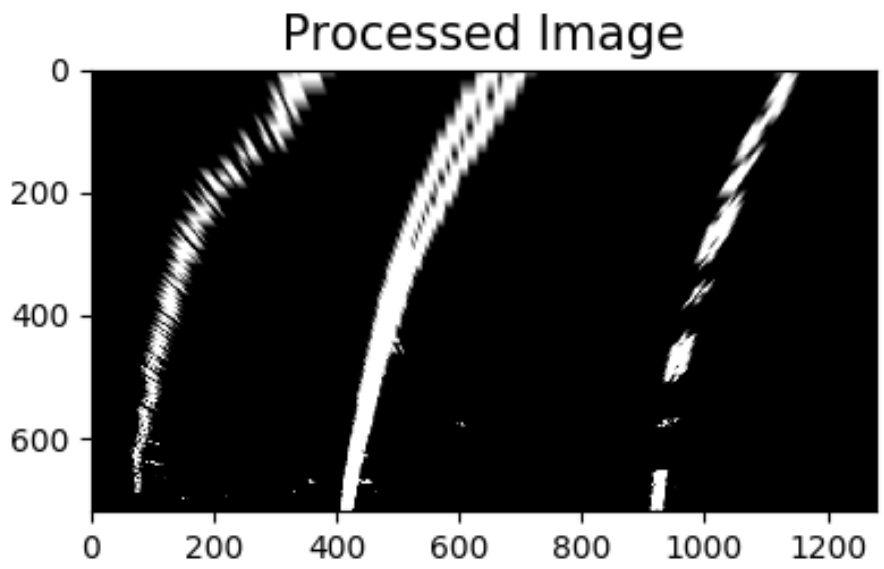


I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

Using relatively straigt images are helpful. Transformation applied to a straight line image:



Transformation applied to a test image:



Lane Segment detection

The code for the lane segment detection is provided in the `LineTracker.py` file. The class `LineTracker` is initialized with the desired convolution window size, search margin, real world conversion coefficients and smoothing factor. The main function that calculates the lane segments is the `find_window_centroids` function. This function first looks at the bottom quarter of the image and finds the maximum convolution result at the left and right side. For convolution it uses a window with ones as a starter. After finding the first centers, it uses this information and goes up in the image in 9 equal intervals (which can be adjusted by changing the window height) applying convolution with the image segments and previously found centers (with windows around). At each level convolution is applied in a region defined by the previous center and margin defined since we know that lane lines are continuous so there is no reason to move away from the previously found centers if there is not a significant/sharp turn.

Lane line centers are added to the `window_centroids` list at each level and stored in the class. From frame to frame there can be slight/moderate differences in the calculated lane segment positions. To get a smoother video with superimposed lane lines, it is better to smooth lane positions using historical data. `smooth_factor` defines the degree of smoothing.

```
    # the main tracking function for finding and storing lane segment positions
    def find_window_centroids(self, warped):

        width = self.__window_width
        height = self.__window_height
        margin = self.__margin

        # store the (left, right) window centroid positions per level
        window_centroids = []
        window = np.ones(width)  # Create our window template that we will use for c
onvolutions

        # first find the two starting positions for the left and right lane by using
to get the vertical image slide
        # and then np.convolve the vertical image slide with the window template
```

```
        # sum the bottom quarter of the image
        # first do it for the left side of the bottom quarter, sum for each column
        l_sum = np.sum(warped[int(3 * warped.shape[0] / 4):, :int(warped.shape[1] /
2)], axis=0)
        # find the maximum value after convolution with the window which is all ones
        # the peak will be a plateau with a length of window width and argmax will f
ind the rightmost point
        # subtract half of the window width to get the center;
        l_center = np.argmax(np.convolve(window, l_sum)) - width/2
        # do the summation for the right side of the bottom quarter of the image
        r_sum = np.sum(warped[int(3 * warped.shape[0] / 4):, int(warped.shape[1] / 2
):], axis=0)
        # now after subtracting half of the window width add the first half of the p
osition
        r_center = np.argmax(np.convolve(window, r_sum)) - width / 2 + int(warped.sh
ape[1] / 2)

        # Add what we found for the first layer
        window_centroids.append((l_center, r_center))

        # Go through each layer looking for max pixel locations
        for level in range(1, (int)(warped.shape[0]/height)):

            # convolve the image into the vertical slice of the image
            img_layer = np.sum(warped[int(warped.shape[0] - (level + 1) * height):in
t(warped.shape[0] - level * height), :],
                               axis=0)
            conv_signal = np.convolve(window, img_layer)


            # find the best left centroid by using past left center as a reference
            # use width/2 as offset because convolution signal reference is at right
side of window, not center ofwindow
            # look only to the left and right side of the previous maximum

            offset = width / 2
            l_min_index = int(max(l_center + offset - margin, 0))
            l_max_index = int(min(l_center + offset + margin, warped.shape[1]))
            l_center = np.argmax(conv_signal[l_min_index:l_max_index]) + l_min_index
- offset

            # Find the best right centroid by using past right center as a reference
            r_min_index = int(max(r_center + offset - margin, 0))
            r_max_index = int(min(r_center + offset + margin, warped.shape[1]))
            r_center = np.argmax(conv_signal[r_min_index:r_max_index]) + r_min_index
- offset

            # Add what we found for that layer
            window_centroids.append((l_center, r_center))

        self.__recent_centers.append(window_centroids)
```

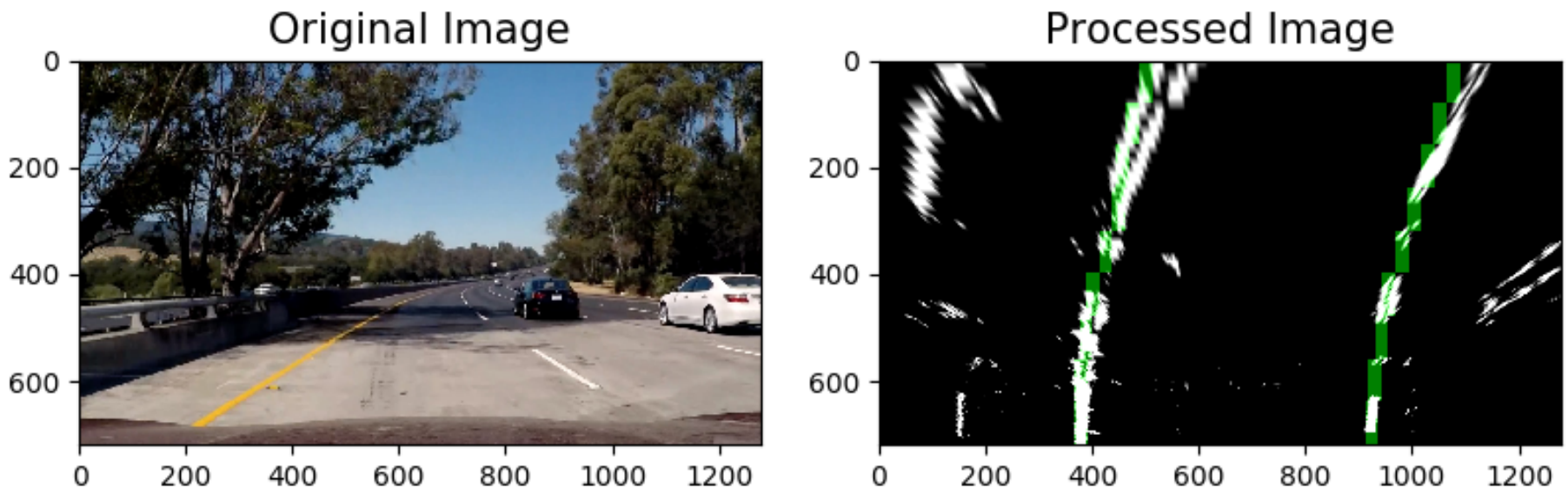
```
        # return average values of the line centers, helps to keep the markers from
jumping around too much
        # let's look at past 15 values (default for smooth factor)
        return np.average(self.__recent_centers[-self.__smooth_factor:], axis=0)
```

The class is initialized in the main `pipeline` function in `flib.py` at line 429. For the test video, these values are used:

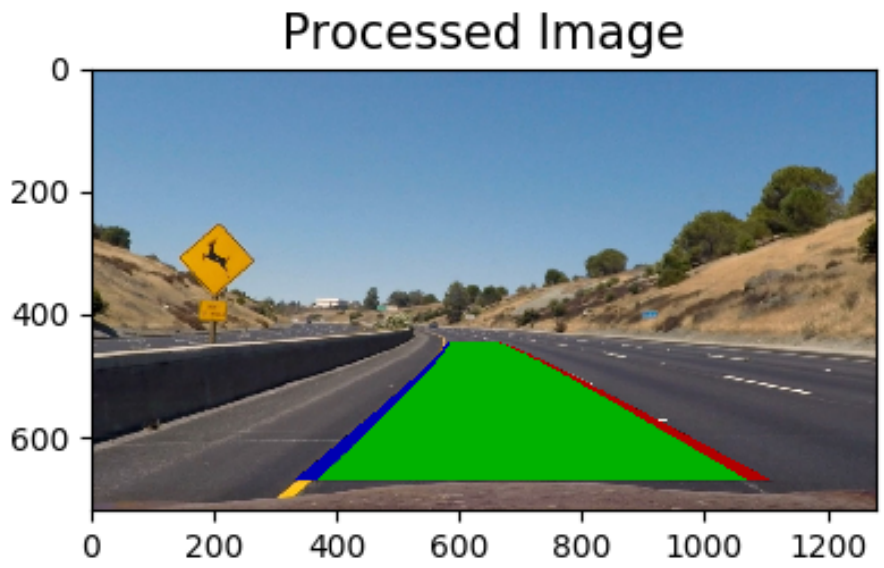
Settings	Value
window_width	25
window_height	80
margin	25
smoothing_factor	15
y_scale	10./720
x_scale	4./384

`y_scale` and `x_scale` are the values to convert pixel values into real world meters. x_scale depends on the `offset` value used in the perspective transformation.

After we find the centroids we can draw these on the image with windows around them. `leftx` and `rightx` lists contains the horizontal (x) pixel position of the centroids. `lmask` and `rmask` will use the `window_mask` function that just pads the given centroid with a window around it. We can add these windows on the orginal image using `cv2.addWeighted()` function. Here is a case applied to a test image. Green windows on the image shows that we succesfully found the lane lines.



Next, we can convert the segments to smooth lines using polynomial line fitting. This section is coded starting at line 476 in `flib.py` file. `yvals` list contains all the vertical positions with 1px increments. `res_yvals` are the centorids we found. We only have 9 values. Using the `leftx` and `res_yvals` we can get a second order polynomial fit and use it to predict the x pixel values for each y point in `yvals` list. `left_lane` and `right_lane` are the lane lines after adding depth (window size) to make them stand out. `inner_lane` is the area between the lines. Using `cv2.fillPoly()` function we can add them to an empty array and then superimpose them on the road image. Using the inverse transform matrix `mat_inv` , we can convert the image back to regular view.



Offset of the car from the center line and road radius of curvature

Offset of the car from the center is calculated based on the x pixel values that are closest to the car. The middle point between the left and right fitted lines at the bottom represents the lane center. If the center of the binary warped image is smaller than this value, the car is on the left of the center; otherwise it is positioned to the right of the center line. The section that calculates the offset is in `flib.py` file between lines 547 and 556.

The road curvature is calculated using the warped images (birds-eye-view images). The pixel values can be converted into real world meter values and the radius of curvature can be calculated based on the bottom of the left line. Since left line in the test video is more robust througout the video. The formula for radius of curvature is based on the circle that is tangent to the line at that point. The equation is defined as follows:

$$R_{curve} = \frac{[1 + (dy/dx)^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

if

$$f(y) = A^2x + Bx + C$$

then

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

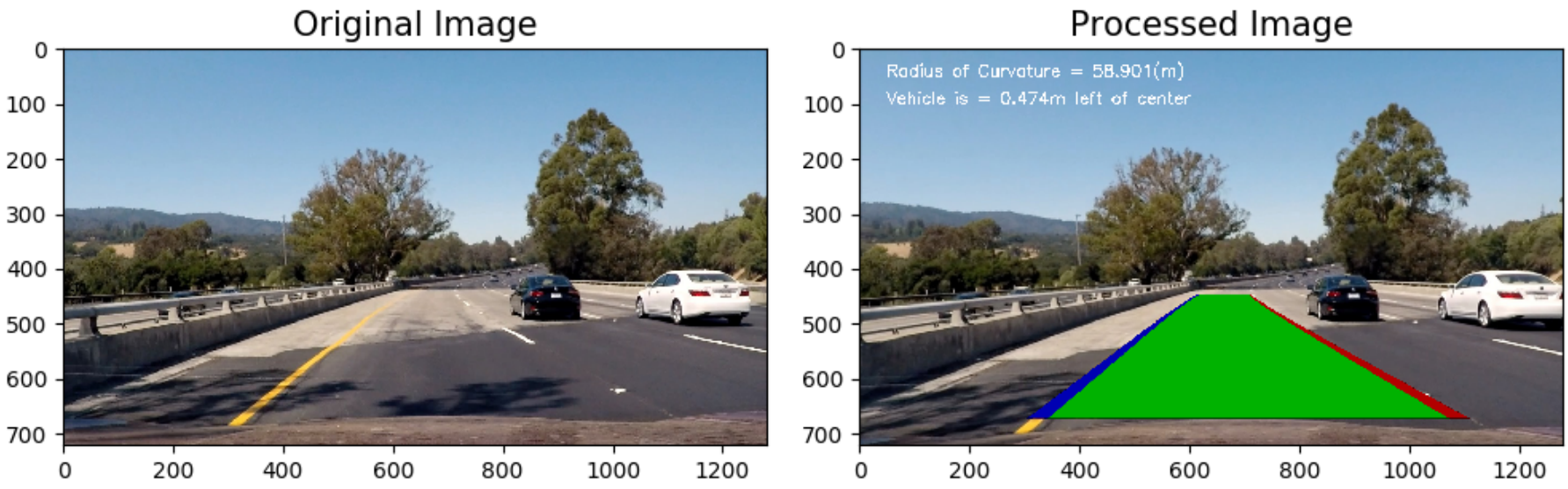
pluggin these into equation of curvature, we get

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

The numerical calculation for this part is again given in the main `pipeline()` function in `flib.py` (540-545).

```
# meters per pixel in y direction
xm_ppx, ym_ppx = centorids.get_ppx_values()
# print(xm_ppx, ym_ppx)

# fit a 2nd order polynomial for the actual x and y coordinates of the left lane
# left lane is more stable
curve_fit_cr = np.polyfit(np.array(res_yvals, np.float32) * ym_ppx, np.array(leftx,
np.float32) * xm_ppx, 2)
# using the formula calculate the road curvature
curverad = ((1 + (2 * curve_fit_cr[0] * yvals[-1] * ym_ppx + curve_fit_cr[1]) ** 2)
** 1.5) / np.absolute(2 * curve_fit_cr[0])
# print(curverad)

# calculate the offset of the car on the road
# average the x pixel values that are closest to the car to find the road center
road_center = (left_fitx[-1] + right_fitx[-1]) / 2
# find the difference between the road center and the warped image center - convert
it to actual meters
center_diff = (road_center - binary_warped.shape[1]/2) * xm_ppx
side_pos = "left"
if center_diff <= 0:
    # if difference is smaller than zero, warped image center (and hence the car) lo
cation
    # is to the right of the road
    side_pos = "right"

# draw the text showing curvature, offset and speed
cv2.putText(output, 'Radius of Curvature = ' + str(round(curverad, 3)) + '(m)', (50,
50),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2)
cv2.putText(output, 'Vehicle is = ' + str(abs(round(center_diff, 3))) + 'm ' + side_
pos + ' of center',
            (50, 100), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2)
```

Here is the image with the radius of curvature and offset from centerline information added on it.

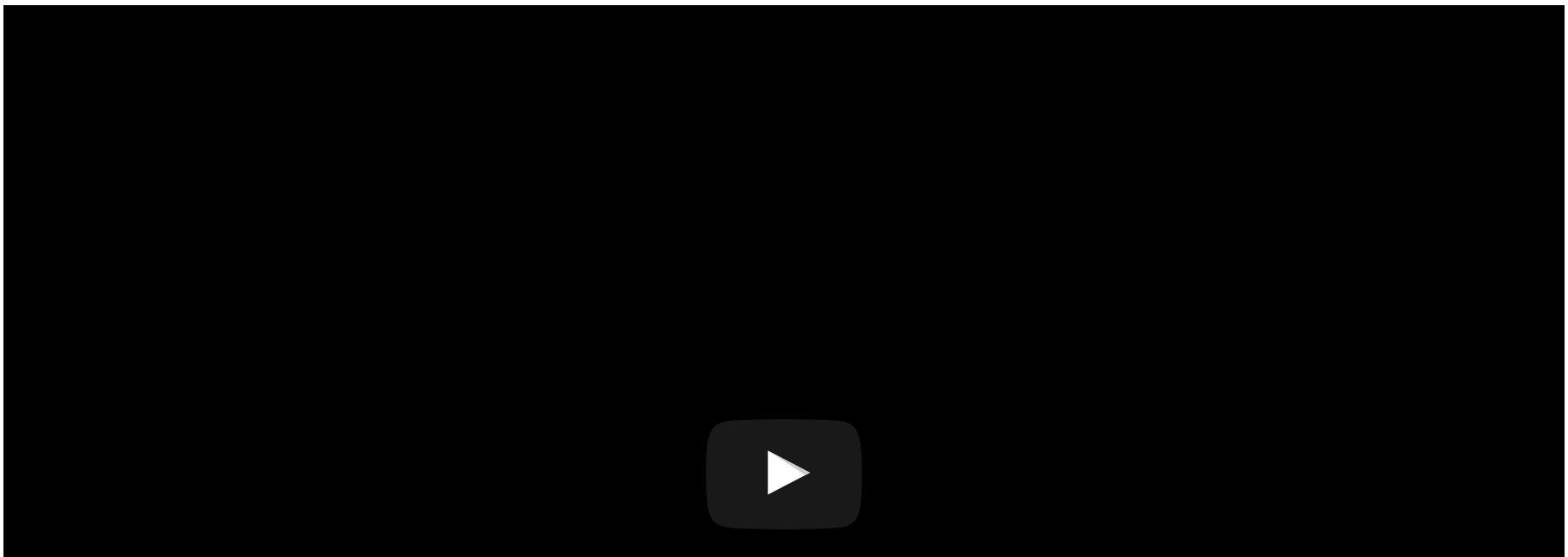


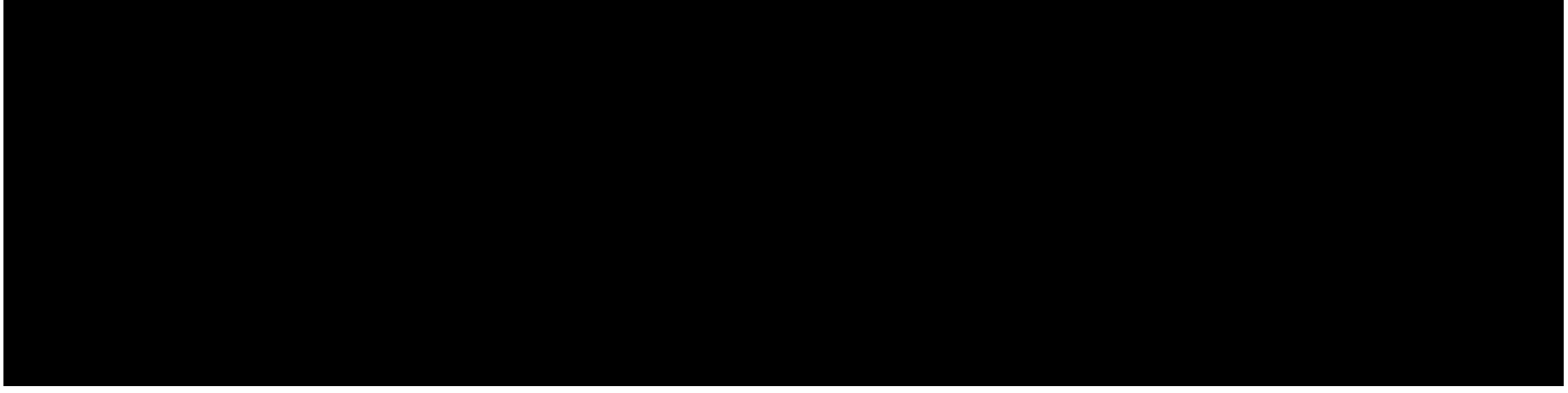
Test Video

To process video files we use `moviepy` package. The code is provided at the bottom of the `flib.py` function.

```
def process_video(file_path, out_folder):

    input_video = file_path
    output_video = os.path.split(os.path.splitext(file_path)[0])[1]
    output_video = os.path.join(out_folder, str(output_video + '_processed.mp4'))

    clip1 = VideoFileClip(input_video)
    video_clip = clip1.fl_image(pipeline)
    video_clip.write_videofile(output_video, audio=False)
```

The output video.





Discussion

Although the pipeline described above works for the majority of the video, there are some sections where it breaks momentarily which indicates that it needs to be improved further. Also testing the pipeline using other videos could be a great way to understand how flexible it is. Some improvements could be listed as:

- Faster image processing by optimizing the pipeline and reducing the search areas.
- Using the lane line found in previous image and defining a region around them for searching.
- Checking for parallelism and assigning weights for lines that are closer to previously found line to smooth out or replace broken lines (left to right adjustment)
- Use an algorithm to do perspective transform

Note: I'll continue to work on improving the model and update this post.