Objective

The objective of this project is to develop an algorithm to find lanes in images and videos taken from a vehicle camera. The algorithm in this project is expected to be of advanced complexity as compared to Project 1 and is expected to asccount for factors such as curving roads.

Approach

The following are the key steps in achieve lane detection:

1. Camera calibration
2. Remove distortion in images
3. Leverage color transforms and image binarization techniques to find lane edges in the image
4. Apply perspective transform
5. Detect lane lines
6. Determine radius of curvature of the road and distance from the center
7. Warp the detected lane boundaries back onto the original image

Since videos are a continuous set of images in the form of frames, the logic applied to detect lanes in video frames is modified to leverage the knowledge of lanes detected in the previous frame

Project Structure

There are two key folders for this project (the necessary README files have been placed in the respective folders):

1. **project** : This folder contains scripts related to this project

    - image_utils.py : This module contains all the image utility functions such as image sharpening, camera calibration, undistortion, binary image creation, perspective transformations, filling polygons and creating final frames
    - lane_utlis.py : This module contains all the lane utility functions such as finding the lane lines, fiting a polynomial function to the lane points, measuring the radius of curvature and distance from centre
    - project2_advanced_lane_finding.ipynb : This is the main notebook which contains the pipeline definitions for image and video lane finding algorithms. A pipeline typically calls in various image and lane finding utility functions to detect lanes in images or videos

2. **output_images** : This folder contains the images generated at various points in the lane detection pipelines.The outputs at different stages in the pipeline are saved for the image only files. For video inputs, only the final video is saved

    - camera_cal : This folder contains the output of camera calibrated chessboard images
    - undistorted_images : This folder contains the undistorted images of the 8 test images
    - binary_images : This folder contains the binary images of the test images
    - warped_images : This folder contains the perspective transformed binary test images (prefix binary_*), perspective transform of the undistorted non binary image with the perspective polygon (prefix warped_*) and undistorted image with the perspective polygon (inverse perspective transformed; prefic undistorted_*) of the test images
    - warped_lanes : This folder contains the perspectie trasnformed bianry images with a polynomial line fit for the lanes along with the sliding windows that aid the lane detection

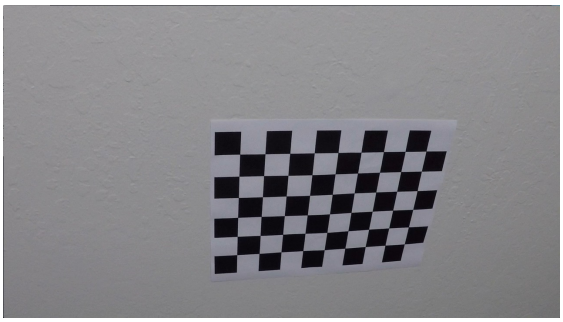
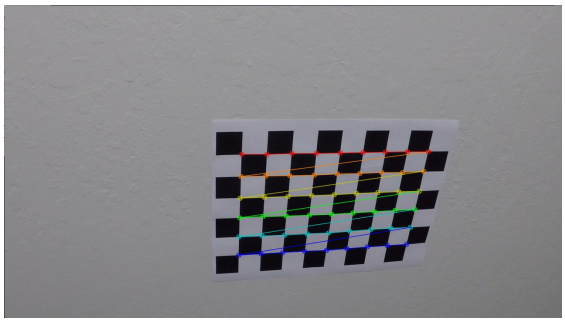
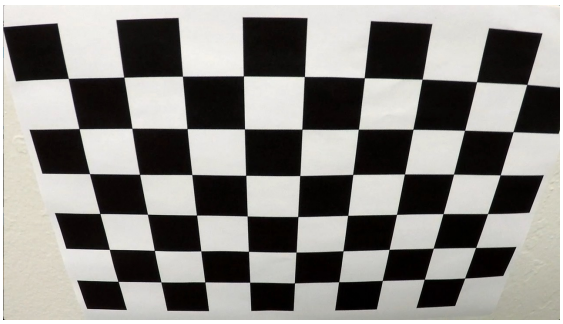
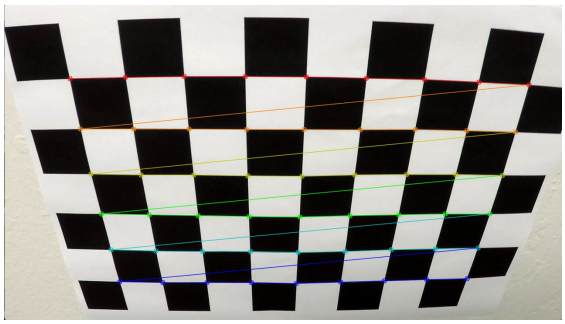
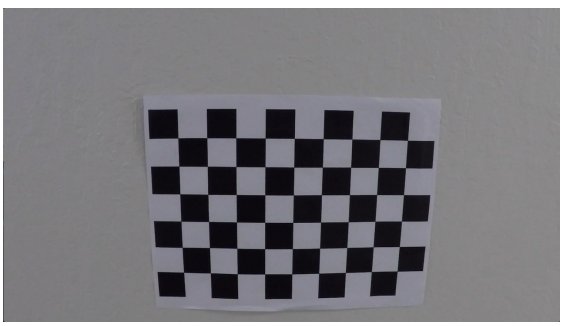
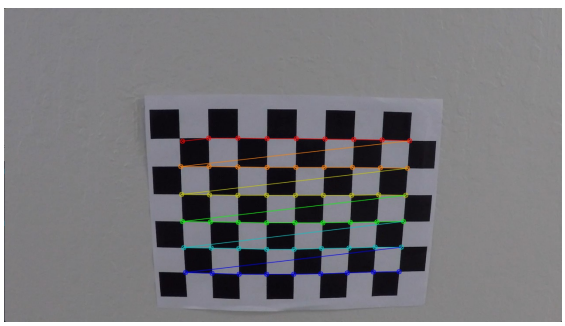
- final_frames : This folder contains the final output images with the detected lanes, radius of curvature and distance from center
- vidoes : This folder contains the vidoe outputs for the project videos

Lane detection pipeline

The pipelines for image lane detection and video lane detection differ in the sense that the video lane detection pipeline leverages previously detected lane information (from previous video frames). Both the pipelines require as input camera calibration parameters to undistort the images / frames.

Camera Calibration

Camera calibration is perfromed using the provided chessboard images. Some sample uncalibrated - calibrated combinations are as follows:

Lane detection pipeline steps

The following function summarizes the lane detection for single images

NOTE : Please refer to the Jupyter notebook for details on all function arguments. The purpose of all code snippets in this document is to understand the flow of logic and not necessarily all the arguments

In [ ]:

```
def image_lane_finding_pipeline(image, write_file = False, file_name = None):
    # 1. Undistortion
    undistorted_image = image_undistortion(image, nx, ny, camera_calibration_params,

    # 1.a. Sharpen image
    sharpened_image = image_sharpen(undistorted_image)

    # 2. Binary Image
    binary_image = create_binary_image(sharpened_image, mask_points, write_file, fil

    # 3. Perspective transform
    warpped_image_binary = create_or_inverse_perspective(binary_image, source_points
    warpped_image = create_or_inverse_perspective(undistorted_image, source_points,
    undistorted_image_polygon = draw_polygon(undistorted_image, source_points,
                                             True, "undistorted_" + file_name.split(
    warpped_image_polygon = draw_polygon(warpped_image, destination_points,
                                         True, "warped_" + file_name.split('/')[-1])

    # 4. Fit polynomial
    image_lanes, image_lanes_plotted, ploty, left_fit, \
    right_fit, leftx_base, rightx_base = fit_polynomial(warpped_image_binary, write_
                                                                                  wr

    # 5. Compute radii and distance to center
    imshape = binary_image.shape
    radii = measure_curvature_real(ploty,ym_per_pix,xm_per_pix, left_fit, right_fit)
    dist_to_center = measure_distance_to_center(imshape, ploty, ym_per_pix, xm_per_p

    # 6.a. Inverse perspective
    unwarped_image = fill_poly(undistorted_image, warpped_image_binary, ploty, left_
                               source_points, destination_points)

    # 6.b. Add radii and distance and save file
    final_frame = create_final_frame(unwarped_image, radii, dist_to_center, True, fi

    results = final_frame.copy()
    return results
```

In case the input to the pipeline are frames from a video file, the pipeline looks like this

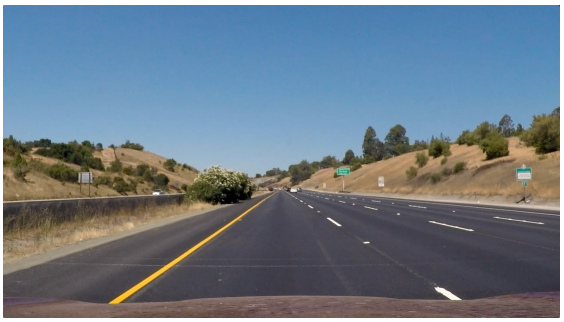
In [ ]:

```
def video_lane_finding_pipeline(image, leftx_base=None, rightx_base=None, ploty = No
                                left_fit = None, right_fit = None,
                                write_file = False, file_name = None):
    # 1. Undistortion
    undistorted_image = image_undistortion(image, nx, ny, camera_calibration_params)

    # 1.a. Sharpen image
    sharpened_image = image_sharpen(undistorted_image)

    # 1.b. Smooth image
    smooth_image = cv2.GaussianBlur(sharpened_image, (5, 5), 1.0)

    # 2. Binary Image
    binary_image = create_binary_image(smooth_image, mask_points, write_file, file_n

    # 3. Perspective transform
    wrapped_image_binary = create_or_inverse_perspective(binary_image, source_points
    wrapped_image = create_or_inverse_perspective(undistorted_image, source_points,
    undistorted_image_polygon = draw_polygon(undistorted_image, source_points)
    wrapped_image_polygon = draw_polygon(wrapped_image, destination_points)

    # 4. Fit polynomial
    try:
        image_lanes, image_lanes_plotted, ploty, left_fit, right_fit, \
    leftx_base, rightx_base = fit_polynomial(wrapped_image_binary, leftx_base, right
    except:
        return undistorted_image

    # 5. Compute radii and distance to center
    imshape = binary_image.shape
    radii = measure_curvature_real(ploty,ym_per_pix,xm_per_pix, left_fit, right_fit)
    dist_to_center = measure_distance_to_center(imshape, ploty, ym_per_pix, xm_per_p

    # 6.a. Inverse perspective
    unwarped_image = fill_poly(undistorted_image, wrapped_image_binary, ploty, left_
                               source_points, destination_points)

    # 6.b. Add radii and distance and save file
    final_frame = create_final_frame(unwarped_image, radii, dist_to_center)

    results = final_frame.copy()
    return (results, leftx_base, rightx_base, ploty, left_fit, right_fit)
```

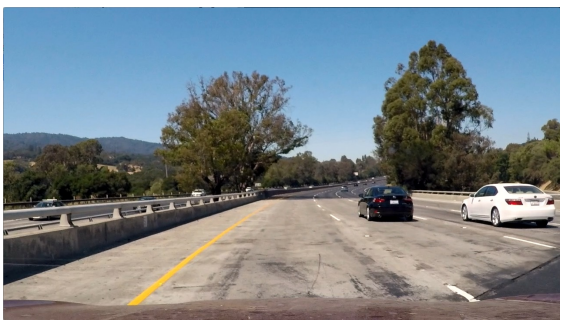
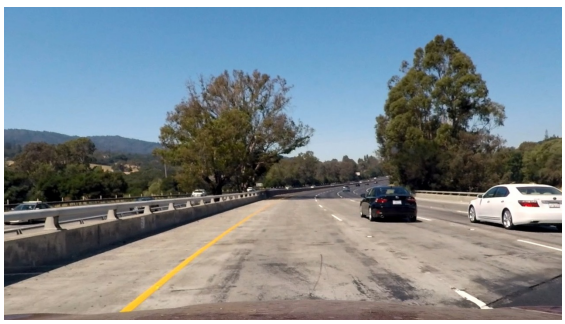
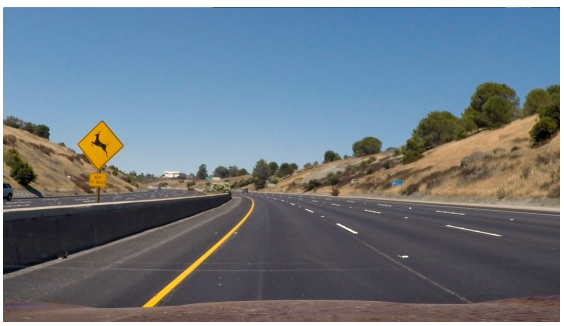
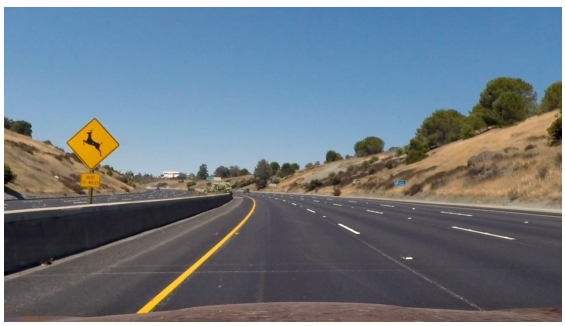
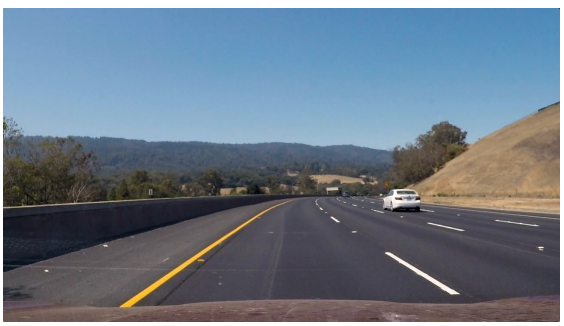
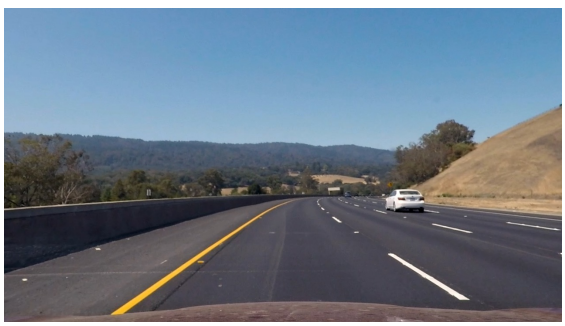
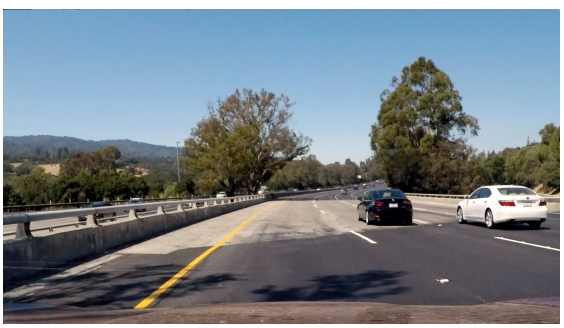
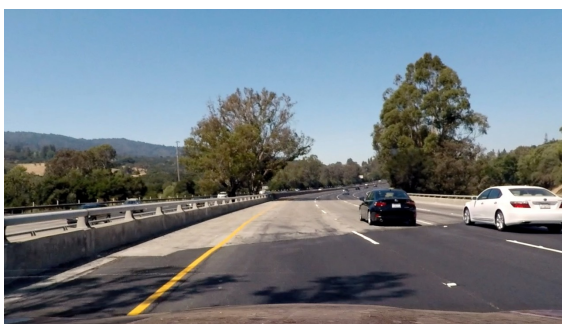
Apart from some differences in the preprocessing steps in Step 1, the key differece is in Step 4 and this difference will be explained below.

Step 1 : Undistortion and preprocessing

This step primarily involves undistorting the image based on the camera calibration parameter. The distorted and undistorted versions of the test images look as below:








Undistortion is followed by some basic processing of images like image sharpening or smoothing - this is dependant on the pipeline and have been found by experimentation

Step 2 : Create binary image

The snippet below summarizes the creation of the binary image. After experimentation, the S and L layers of the image were chosen for application of Sobel filter in the x direction with appropriate thresholds. Other layers experimented were layers in the YCrCb, LAB, LUV, RGB, grayscale color transforms. Sobel y and magnitude / direction were also experimented but not with better results.

The output is the binary image with edges highlighted over a masked area in the image. The masked area corresponds to the region in which the lane is expected to be found.

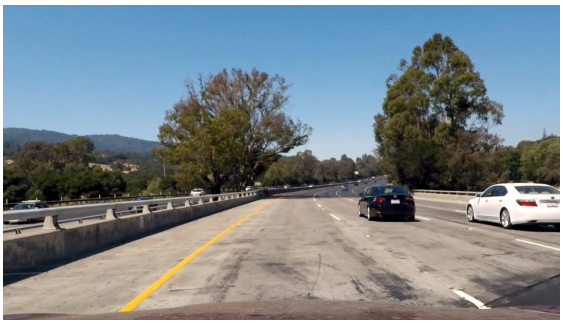
In [ ]:

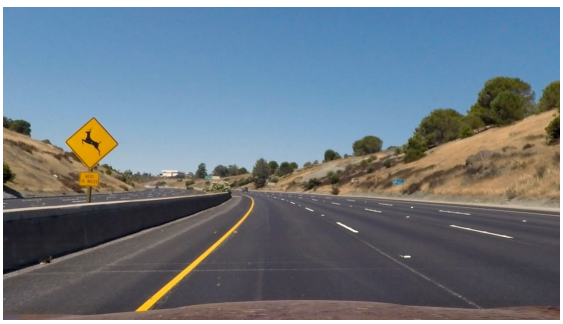
```
hls = cv2.cvtColor(undistorted_image, cv2.COLOR_RGB2HLS)

s_channel = hls[:,:,2]
sobelx = cv2.Sobel(s_channel, cv2.CV_64F, 1, 0, ksize = 5)
abs_sobelx = np.absolute(sobelx)
scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))
sxbinary = np.zeros_like(scaled_sobel)
sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

l_channel = hls[:,:,1]
sobelx = cv2.Sobel(l_channel, cv2.CV_64F, 1, 0, ksize = 5)
abs_sobelx = np.absolute(sobelx)
scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))
lxbinary = np.zeros_like(scaled_sobel)
lxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

combined_binary = np.zeros_like(s_channel)
combined_binary[((sxbinary==1) | (lxbinary==1))] = 1
mask = np.zeros_like(s_channel)
cv2.fillPoly(mask, np.int_([mask_points]), 1) #mask_points define the edges of the r
combined_binary_mask = np.bitwise_and(combined_binary, mask)
```

The binary outputs of the undistorted images look as follows



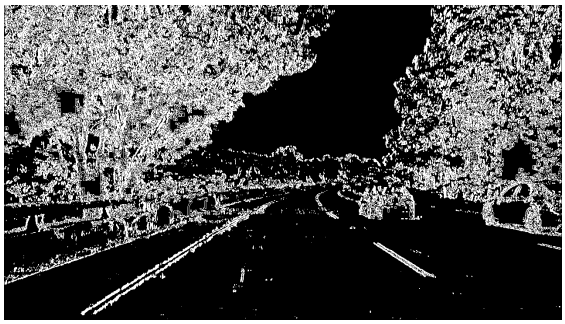








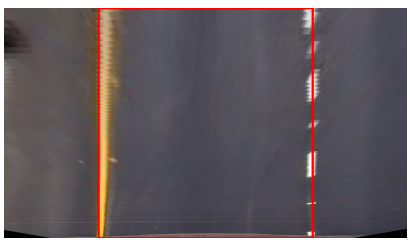
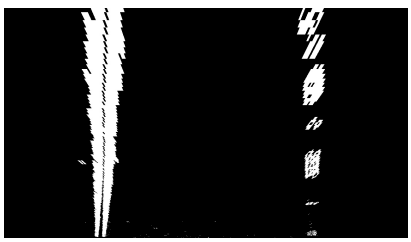


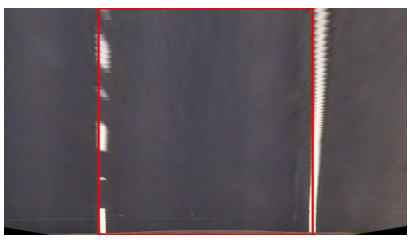
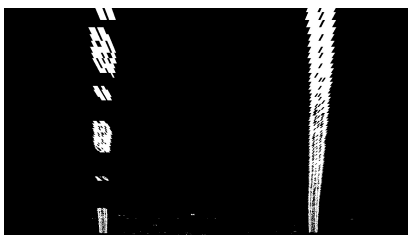


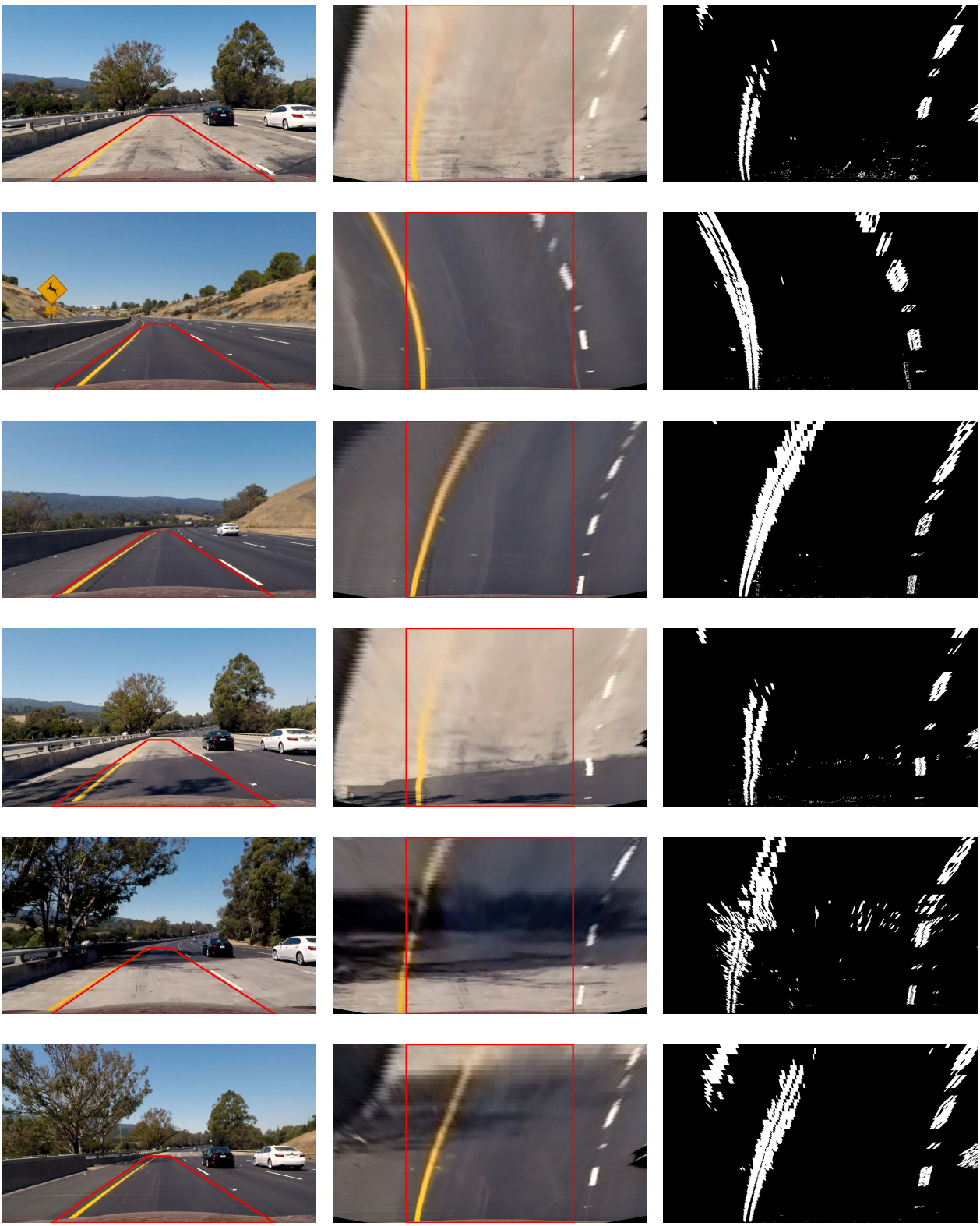
Step 3 : Perspective transform

The binary image is warped into a bird's eye view by perspective trasnform. The source and destination points to define the perspective were defined by experimentation.

The following array of images display the images in the following order:
Undistorted image with perspective polygon -> Perspective polygon on warped image -> Binary version of the warped image



Step 4 : Fit Polynomial

Fit polynomial for lane in an image

The **fit_polynomial()** function in the lane utilities is used for this purpose. This function takes as input the warped binary image and applies the sliding window technique to detect the lane points. Once the lane points are detected, polynomial function is fit to these points to define the lane curve equation.

Modification for image that is a frame in a video

In case the image is a frame in a video, the pipeline preserves the polynomial fit from the the previous frame and passes it onto the **fit_polynomial()** function. This function then leverages this knowledge to compare the polynomial fit obtained from the current image; in case the fit is not defined or defined on the basis of **points less than a threshold value**, the previous polynomial is continued in the current image as well. The threshold is preserved by the **min_pix_replot** variable. The following code snippet exhibits the logic to obtain this comparison, the **find_lane_pixels()** function is the function that deploys the sliding window technique to identify the lane pixels. Apart from the polynomial fit, the **fit_polynomial()** function also preserves the starting points of the previously detected lanes to be used by the sliding window algorithm as explained in the next note.

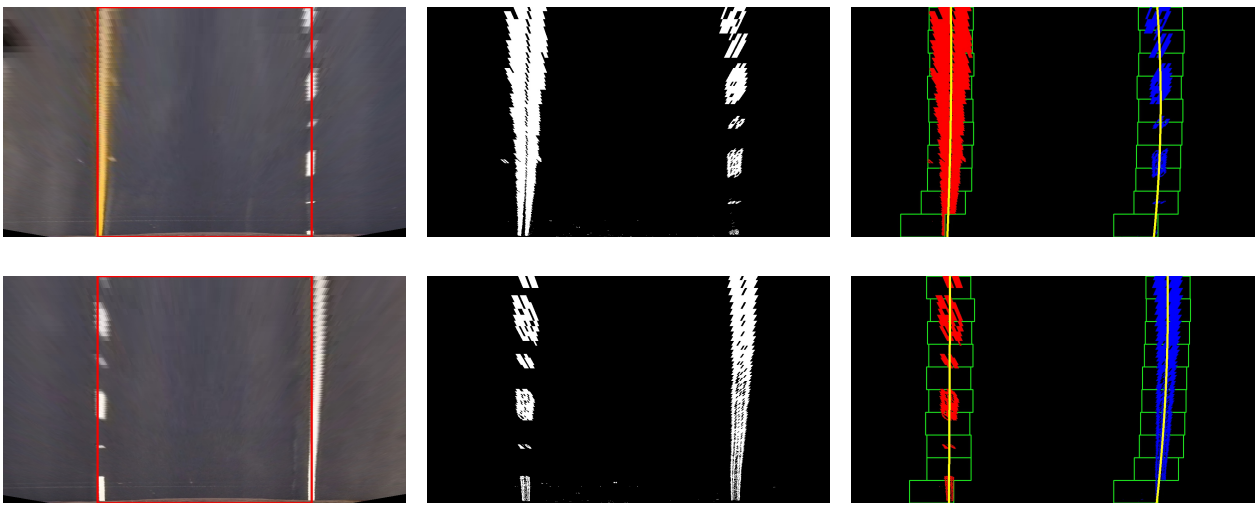
Note on sliding windows

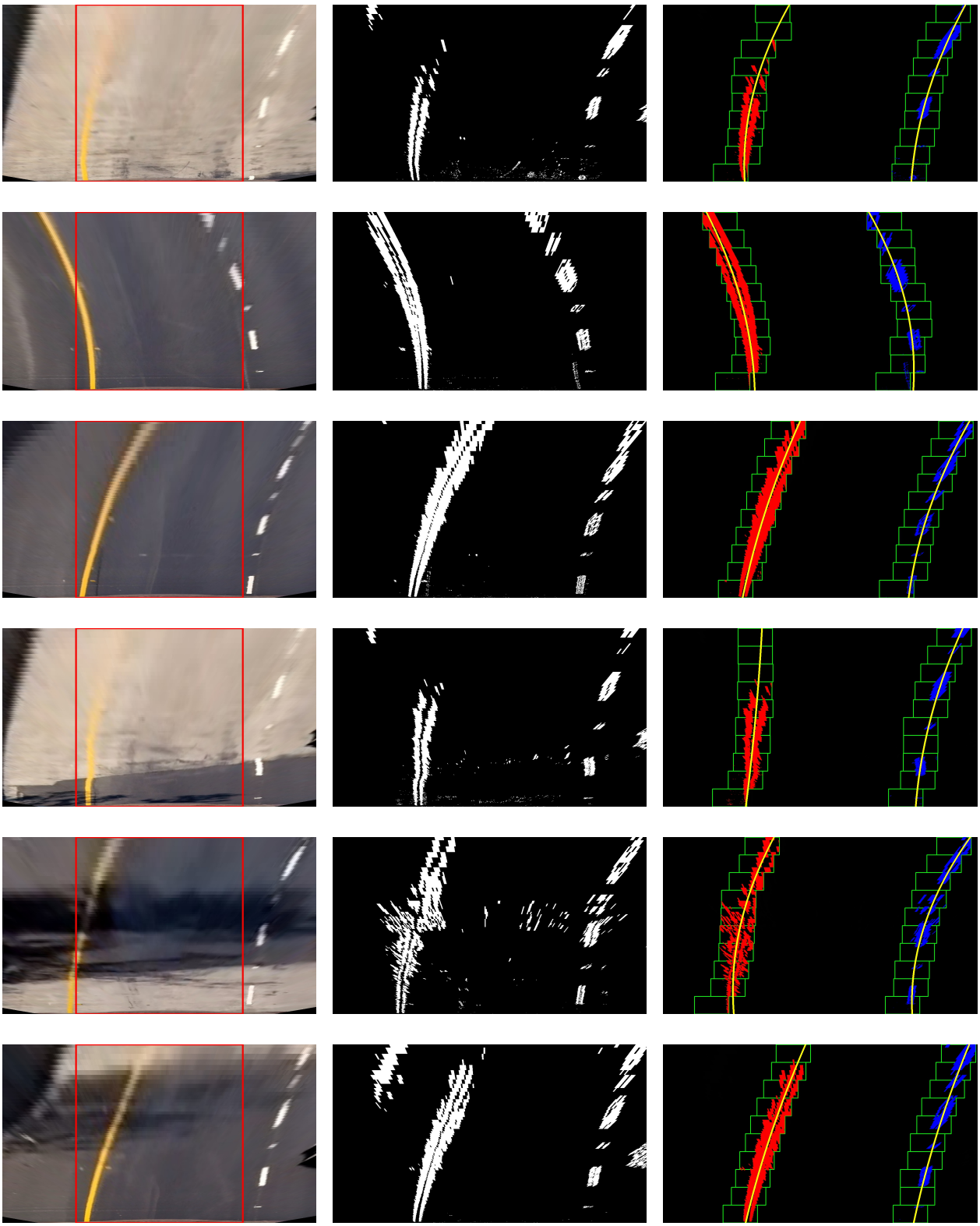
- The starting point for the left lane sliding windows and the right lane sliding windows is determined by the pixel density (determined by the histogram of the lower half of the image) across the x axis OR as passed on from the previous frame of the video.
- Once pixels are detected in a window the position of the new window is determined by the median x-values of the locations of the detected pixel. The median is chosen to account for outliers that might influence the standard mean computation.

In [ ]:

```
min_pix_replot = 8000
leftx, lefty, rightx, righty, out_img, leftx_base, rightx_base = find_lane_pixels(bi

# At this point left_fit holds the previous image polynomial, leftx holds the newly
# Similar for the right lane
if ((left_fit==None) | (leftx.shape[0] > min_pix_replot)):
    left_fit = np.polyfit(lefty, leftx, 2)
else:
    pass
if ((right_fit==None)| (rightx.shape[0] > min_pix_replot)):
    right_fit = np.polyfit(righty, rightx, 2)
else:
    pass
```

The following array of images display the images in the following order:
Perspective polygon on warped image -> Binary version of the warped image -> Lane fitted binary perspective with sliding windows





Step 5 : Metric computations

The following two metrics are computed as explained by the code snippets

1. Radii of curvature

In [ ]:

```
def measure_curvature_real(ploty, ym_per_pix, xm_per_pix, left_fit, right_fit):
    """ Calculates the curvature of polynomial functions in meters """
    y_eval = np.max(ploty)
    left_curverad = ((1 + (2*left_fit[0]*y_eval*ym_per_pix + left_fit[1])**2)**1.5)
    right_curverad = ((1 + (2*right_fit[0]*y_eval*ym_per_pix + right_fit[1])**2)**1.

    return (left_curverad+ right_curverad)//2
```

2. Distance of lane center to camera center

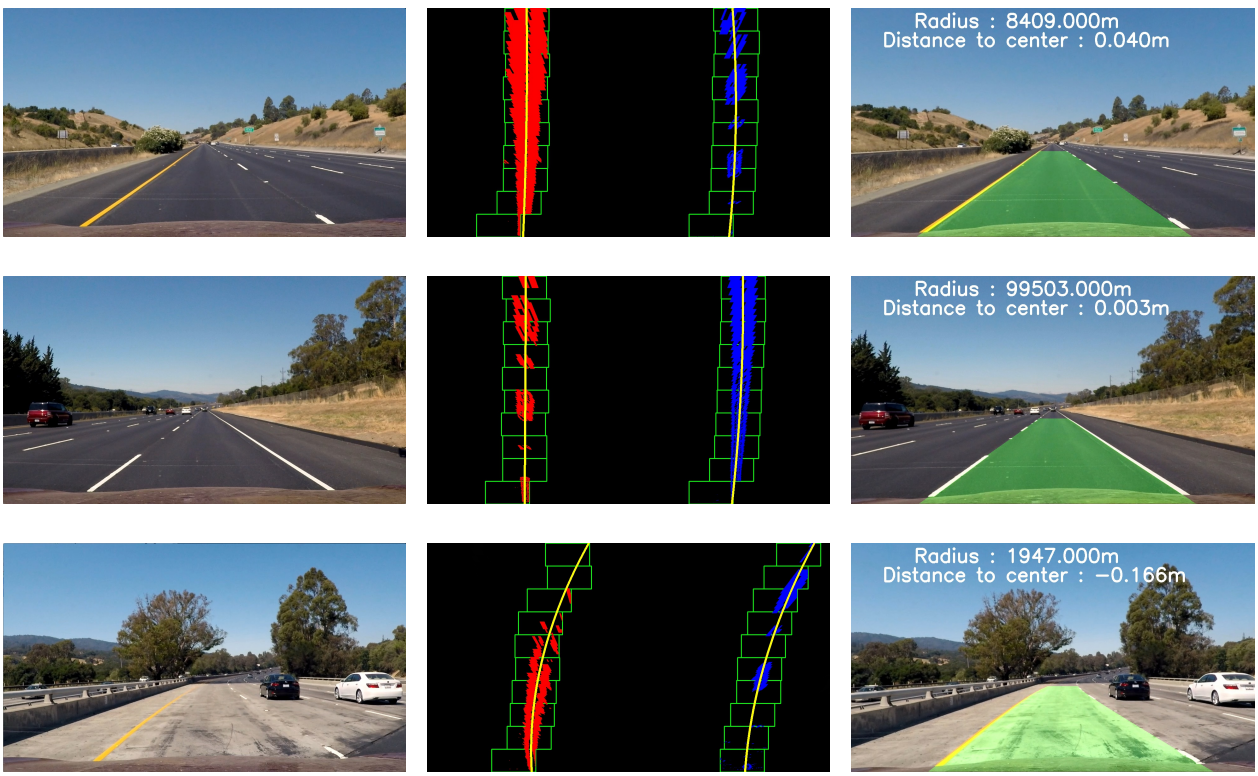
In [ ]:

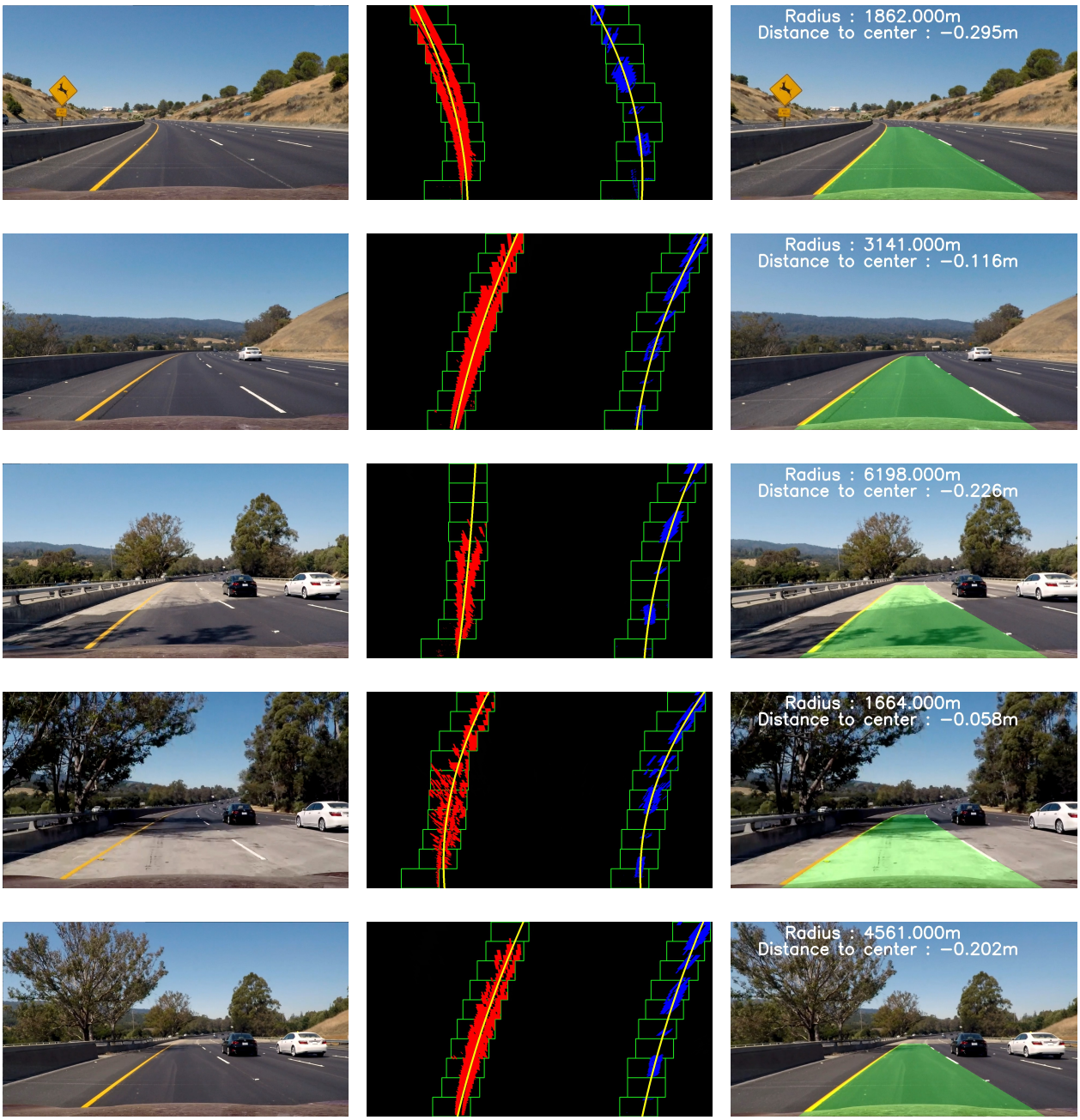
```
def measure_distance_to_center(imshape, ploty, ym_per_pix, xm_per_pix, left_fit, rig
    """ Calculates the distance of camera center to lane center in meters """
    y_eval = np.max(ploty)

    left = np.polyval(left_fit, y_eval)
    right = np.polyval(right_fit, y_eval)
    center = imshape[1]/2
    dist_to_center = (center - (left + right)/2)*xm_per_pix
    return dist_to_center
```

Step 6 : Inverse perspective transformation and unwarping of image

The final step is to perform inverse persepective transform on the binary perspective image with lanes detected to obtain the image with the area between the lanes highlighted and display radius of curvature and distance of center of lanes to camera center.

The following array of images display the images in the following order:
Original Image -> Lane fitted binary perspective with sliding windows -> Final output image





Video lane detection

The approach explained above works well with the project video (see : /output_images/videos/output_project_video.mp4). However it does not quite work well with the challenge videos.

- With the challenge_video.mp4, one of the issue is that of a parallel shade occuring at the seeming point of convergence between the two lanes. The effect of this is seen right from the beginning as the detected region on the left take a left curve rather than a right one. Investigating the binary perspective transformed image of this frame reveals that the top portion of the left lane included edges detected from the road edge (inside the shade). This can potentially be removed by better image pre processing techniques to detect only the lane edges and remove the other edges.
- For the harder_challenge_video.mp4, the issue is shades and changing elevations combined with curves - all happening almost at the same locations. We need better algorithms to account for such changes