1. **Camera Calibration:**
   1. To find the camera calibration matrix and the distortion coefficients, I first made object points and image points. These were for the chessboard images and were made to be 9X6.
   2. Next, I read in all of the chessboard images that were provided and converted them to gray-scale. Once this was done, I used the openCV function findChessBoardCorners. And when this function returned true I was able to get the corners and append them to my object and image point lists
   3. And then I used the calibrateCamera function to get the mtx and dist values that would be used to undistort the images throughout this project
   4. Finally, once this was accomplished I grabbed a test image of a chessboard and undistorted it. This image is stored in the output\_images file under undistorted.jpg
2. **Threshold Binary Images:**
   1. Once I had everything that I needed from the chessboard images it was time to retrieve the actual test images.
   2. The first step here was to figure out a robust way to turn these images into binary images that could later be used to extract lane lines. Step one was to undistort the incoming image.
   3. After this I decided to combine two different methods to achieve a better binary image in the end. To accomplish this I combined an HLS binary image which used a threshold of (170, 255), and a sobel X operator with threshold values of (20, 100). This seemed to work rather well in finding the pixels that were most important, however it also picked up many pixels that weren't as important, which would be dealt with during the perspective transformation step.
   4. There are three examples of this ‘to\_binary’ function at work, and they can all be found in the output\_images folder under binary1, binary2, and binary3.
3. **Perspective Transformation:**
   1. I found this step to be somewhat challenging and I would imagine by tweaking this aspect of my project I could achieve even better results.
   2. I first spent some time playing around with different values so that I could better understand the functions being used here. And I would test them out on the straight road images until I was happy with the results
   3. From there I defined a ‘perspective\_transform’ function to be used later in my pipeline. I changed my values around a bit and ended up with some good outcomes here, despite the amount of tinkering it all required. I also chose to calculate the Minv variable here, to be used later when drawing the lane lines and warping them back onto the original image
   4. Examples of my perspective\_transform function being used on my test binary images are saved as transformed\_binary1, and transformed\_binary2 as jpg files
4. **Detecting Lane Lines:**
   1. This function turned out to be a rather long one so I will try to tackle it from top to bottom without missing too much
   2. Firstly, the incoming image to this function would be a transformed binary image where the lanes are being looked down upon from a birds-eye view.
   3. Starting off, I made a histogram of the columns of the image to see where the binary pixels where most prevalent. To do this for both the left and right column I had to define a midpoint of the histogram and find the X index of the greatest value on both the left side of this midpoint and the right.
   4. Once this was done, I defined the number of vertical windows that would be used to search for the left and right lanes, as well as the height of each of these windows. I also got the X and Y indices of all the nonzero pixels in the original image.
   5. A leftx\_current and rightx\_current variable were also defined so that the windows would have a good starting position to search for lane lines, here is where a margin and minimum pixels count variable were defined as well
   6. Now to start the actual search for the lanes on either side I looped through the number of windows that I knew that would have, this would act as a sort of focused search starting at the bottom of the image and then moving up and following the lane lines. The first thing that needed to be done here was to mark where the corners of the windows would be. This used the margin variable that was already chosen before hand to get the length of each search window, and the window\_height variable to get the height of the search windows. These were then drawn onto the original transformed binary image to visualize the results at the end.
   7. From this point, I grabbed all of the X indices that fell within each of the windows, and once these were found, I appended them to left and right lane indices variables accordingly. I then checked to make sure that a certain amount of pixels where found in each window for each of the two lane lines before moving the center X value of the windows that would be used during the next iteration through the loop. This made it so that by the end of the loop each set of windows would follow their lane lines from bottom of the image to the top.
   8. Once the loop was finished, I then concatenated all of the X indices for the left and right lane lines separately to get one long list of leftx indices and rightx indices. Using these indices I was able to use the nonzerox and nonzeroy lists that were found earlier. By doing this, I was left with four lists. The first two of these lists corresponded to the x and y values of the pixels that were inside of the sliding search windows for the left side of the lane; and the other two corresponded to the same values for the right side.
   9. Finally, using these lists I could fit a 2nd degree polynomial to the points to get line coefficients and then by using these coefficients along with predetermined y values based on the total height of the image, I was able to get the values for a line.
   10. I thought I would also mention the return values of this function since there are a few.
       1. Ploty: the y values of each line, they are the same for each line and were found using the height of the original image.
       2. left\_fitx and right\_fitx: these are each side of the lanes x values that pair up with the y values in ploty to form each individual line.
       3. Leftx\_base and rightx\_base: these are the two starting values of the centers of the sliding search windows. These values are then used later to determine the distance the car is from the center of the road.
       4. Out\_img: this is the original image with the sliding windows drawn on for visualization purposes
   11. To visualize how this function works, you can look under ‘output\_images/detected\_lane.jpg’ Here I used the same test image from perspective\_transform to test the lane detection. It has the original transformed image with green boxes drawn on to represent the sliding windows as well as the final detected lanes drawn in yellow
5. **Finding Curvature and Distance from the center of the lane:**
   1. Here is where I implemented a function that would return the curvature of the lane in meters and the distance the car is from the center of the lane in meters as well
   2. I used the rightx\_base and the leftx\_base variables to find the number of pixels that were in between the left side of the lane and the right. From here I determined how many meters each pixel was representing by taking the standard width of a highway lane into account and dividing by the number of pixels; I did a similar thing in the y direction, however, I am not nearly as confident with that measurement and some guess work was needed here.
   3. Then using these new values I fit another polynomial to these lines, except this time accounting for the meters per pixel values that were just calculated. And then from here I found the actual curvature of each side of the lane and the averaged them to get a more reliable number for the curvature of the lane.
   4. To get the distance the car was from the center of the lane I first found the center of the image which was easy enough to find. I then grabbed the width of the part of the lane that was closest to the car and divided by 2. By adding this value to the leftx\_base I was able to find the middle of the detected lane. I then subtracted this value from the known position of the camera and multiplied that value by the already established meters per pixel value to get the distance from the center of the lane in meters.
6. **Drawing the lanes:**
   1. Drawing the lanes themselves in the end was fairly straight forward. First I created an empty image similar to the original. I then took all the points that were found in the detect\_lanes function and arranged them so that the could better used by the cv2.polyfill function. This was done by putting the x and y values of all of the pixels into a clockwise position essentially, starting at the bottom of the left lane, going up to the top of that lane and then from the top of the right lane down to the bottom closer to the car. Once that was done, I used np.hstack to but all of these points into one array to be fed into the fillpoly function.
   2. I then used these points with fillpoly on the blank image that was created, however, this is all still in warped space. To map this now detected and filled lane to the original undistorted image, I used my Minv value from before and warped this image back to the original image using cv2.warpPerspective() and cv2.addWeighted().
   3. Once this was right, I then proceeded to writing the lane curvature and the distance from the center of the lane onto the top left side of the image
   4. To do this, I converted the undistorted image with lanes drawn onto it from RGB to RGBA. And then used the PIL.Image library to create a similar image to that one and draw each value onto it individually. And finally, I mapped that image onto the lane lines image to produce a final image that was then returned.
   5. You can see these results as ‘result.jpg’ in the output\_images folder provided with this project
7. **Testing the pipeline:**
   1. Finally I wrote and then quickly tested my final pipeline using the function that I have described above. This test has similar results to the one above obviously and is saved as ‘pipeline\_test.jpg’
8. **Video** 
   1. **Applying pipeline to each frame of video:**
      1. I used the cv2.VideoCapture function to apply my pipeline to the video provided by the project.
      2. This is pretty straightforward but i used a while loop to go through the entire video and then for each frame I I changed the image from BGR to RGB and then feed that image along with the matrix and distortion values, which returned the final images. I then saved these images to a data folder.
   2. **Getting video from pipeline images:**
      1. Here I loaded in each image and created a frame\_array list variable. I then sorted each image by its name to make sure they were in chronological order.
      2. From here I looped through the sorted file names, found the files and loaded them in with cv2.imread(), and then appended these images to the frame\_array list.
      3. And finally, I made a VideoWriter object and then looped through the array that I had just made and was able to make a video from these frame arrays. This is provided with my submission as video2.avi
9. **Reflection:**
   1. I thought that this project was very long and required me to learn a lot. And there are definitely some areas where my algorithm could improve. For starters, a smoothing aspect would probably be beneficial here as in the final video the lane jumps for a frame or two in two locations, which could be solved with some type of lane smoothing were the algorithm would ignore such a sudden a dramatic change to the lane. Perhaps along with that, to speed up the pipeline itself I could add in a modified search window that doesn't search each frame completely and can take into account the lane detected in the previous frame.