**Project Steps**

Steps we’ve covered so far:

1. Camera calibration
2. Distortion correction
3. Color/gradient threshold
4. Perspective transform

After doing these steps, you’ll be given two additional steps for the project:

1. Detect lane lines
2. Determine the lane curvature

**Processing Each Image**

A car parked on the side of a road

Description automatically generated

In the project at the end of this module, the first thing you'll do is to compute the camera calibration matrix and distortion coefficients. You only need to compute these once, and then you'll apply them to undistort each new frame. Next, you'll apply thresholds to create a binary image and then apply a perspective transform.

**Thresholding**

You'll want to try out various combinations of color and gradient thresholds to generate a binary image where the lane lines are clearly visible. There's more than one way to achieve a good result, but for example, given the image above, the output you're going for should look something like this:

A picture containing outdoor, photo, holding, skiing

Description automatically generated

**Perspective Transform**

Next, you want to identify four source points for your perspective transform. In this case, you can assume the road is a flat plane. This isn't strictly true, but it can serve as an approximation for this project. You would like to pick four points in a trapezoidal shape (similar to region masking) that would represent a rectangle when looking down on the road from above.

The easiest way to do this is to investigate an image where the lane lines are straight, and find four points lying along the lines that, after perspective transform, make the lines look straight and vertical from a bird's eye view perspective.

**Here's an example of the result you are going for with straight lane lines:**

A picture containing monitor

Description automatically generated

**Now for curved lines**

Those same four source points will now work to transform any image (again, under the assumption that the road is flat and the camera perspective hasn't changed). When applying the transform to new images, the test of whether or not you got the transform correct, is that the lane lines should appear parallel in the warped images, whether they are straight or curved.

Here's an example of applying a perspective transform to your thresholded binary image, using the same source and destination points as above, showing that the curved lines are (more or less) parallel in the transformed image:

A picture containing clock

Description automatically generated

**Locate the Lane Lines**

A picture containing fireworks

Description automatically generated

Thresholded and perspective transformed image

You now have a thresholded warped image and you're ready to map out the lane lines! There are many ways you could go about this, but here's one example of how you might do it:

**Line Finding Method: Peaks in a Histogram**

After applying calibration, thresholding, and a perspective transform to a road image, you should have a binary image where the lane lines stand out clearly. However, you still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

Plotting a histogram of where the binary activations occur across the image is one potential solution for this. In the quiz below, let's take a couple quick steps to create our histogram!

##### Here's the approach I took.

I take a **histogram** along all the columns in the lower half of the image like this:

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

histogram = np.sum(img[img.shape[0]//2:,:], axis=0)

plt.plot(histogram)

The result looks like this:

A close up of a logo

Description automatically generated

## Sliding Window

With this histogram we are adding up the pixel values along each column in the image. In our thresholded binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines. We can use that as a starting point for where to search for the lines. From that point, we can use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

Here is a short animation (no sound!) showing this method:

# Implement Sliding Windows and Fit a Polynomial

A picture containing clock

Description automatically generated

As shown in the previous animation, we can use the two highest peaks from our histogram as a starting point for determining where the lane lines are, and then use sliding windows moving upward in the image (further along the road) to determine where the lane lines go.

### Split the histogram for the two lines

The first step we'll take is to split the histogram into two sides, one for each lane line.

**import** numpy **as** np

**import** cv2

**import** matplotlib.pyplot **as** plt

*# Assuming you have created a warped binary image called "binary\_warped"*

*# Take a histogram of the bottom half of the image*

histogram = np.sum(binary\_warped[binary\_warped.shape[0]//2:,:], axis=0)

*# Create an output image to draw on and visualize the result*

out\_img = np.dstack((binary\_warped, binary\_warped, binary\_warped))\*255

*# Find the peak of the left and right halves of the histogram*

*# These will be the starting point for the left and right lines*

midpoint = np.int(histogram.shape[0]//2)

leftx\_base = np.argmax(histogram[:midpoint])

rightx\_base = np.argmax(histogram[midpoint:]) + midpoint

Note that in the above, we also create out\_img to help with visualizing our output later on.

### Set up windows and window hyperparameters

Our next step is to set a few hyperparameters related to our sliding windows, and set them up to iterate across the binary activations in the image. We have some base hyperparameters below, but don't forget to try out different values in your own implementation to see what works best!

*# HYPERPARAMETERS*

*# Choose the number of sliding windows*

nwindows = 9

*# Set the width of the windows +/- margin*

margin = 100

*# Set minimum number of pixels found to recenter window*

minpix = 50

*# Set height of windows - based on nwindows above and image shape*

window\_height = np.int(binary\_warped.shape[0]//nwindows)

*# Identify the x and y positions of all nonzero (i.e. activated) pixels in the image*

nonzero = binary\_warped.nonzero()

nonzeroy = np.array(nonzero[0])

nonzerox = np.array(nonzero[1])

*# Current positions to be updated later for each window in nwindows*

leftx\_current = leftx\_base

rightx\_current = rightx\_base

*# Create empty lists to receive left and right lane pixel indices*

left\_lane\_inds = []

right\_lane\_inds = []

### Iterate through nwindows to track curvature

Now that we've set up what the windows look like and have a starting point, we'll want to loop for nwindows, with the given window sliding left or right if it finds the mean position of activated pixels within the window to have shifted.

You'll implement this part in the **quiz** below, but here's a few steps to get you started:

1. Loop through each window in nwindows
2. Find the boundaries of our current window. This is based on a combination of the current window's starting point (leftx\_current and rightx\_current), as well as the margin you set in the hyperparameters.
3. Use cv2.rectangle to draw these window boundaries onto our visualization image out\_img. This is required for the quiz, but you can skip this step in practice if you don't need to visualize where the windows are.
4. Now that we know the boundaries of our window, find out which activated pixels from nonzeroy and nonzerox above actually fall into the window.
5. Append these to our lists left\_lane\_inds and right\_lane\_inds.
6. If the number of pixels you found in Step 4 are greater than your hyperparameter minpix, re-center our window (i.e. leftx\_current or rightx\_current) based on the mean position of these pixels.

### Fit a polynomial

Now that we have found all our pixels belonging to each line through the sliding window method, it's time to fit a polynomial to the line. First, we have a couple small steps to ready our pixels.

*# Concatenate the arrays of indices (previously was a list of lists of pixels)*

left\_lane\_inds = np.concatenate(left\_lane\_inds)

right\_lane\_inds = np.concatenate(right\_lane\_inds)

*# Extract left and right line pixel positions*

leftx = nonzerox[left\_lane\_inds]

lefty = nonzeroy[left\_lane\_inds]

rightx = nonzerox[right\_lane\_inds]

righty = nonzeroy[right\_lane\_inds]

We'll let you implement the function for the polynomial in the **quiz** below using np.polyfit.

*# Assuming we have `left\_fit` and `right\_fit` from `np.polyfit` before*

*# Generate x and y values for plotting*

ploty = np.linspace(0, binary\_warped.shape[0]-1, binary\_warped.shape[0])

left\_fitx = left\_fit[0]\*ploty\*\*2 + left\_fit[1]\*ploty + left\_fit[2]

right\_fitx = right\_fit[0]\*ploty\*\*2 + right\_fit[1]\*ploty + right\_fit[2]

Take note of how we fit the lines above - while normally you calculate a y-value for a given x, here we do the opposite. Why? Because we expect our lane lines to be (mostly) vertically-oriented.

### Visualization

Once you reach this point, you're done! But here is how you can visualize the result as well:

out\_img[lefty, leftx] = [255, 0, 0]

out\_img[righty, rightx] = [0, 0, 255]

plt.imshow(out\_img)

plt.plot(left\_fitx, ploty, color='yellow')

plt.plot(right\_fitx, ploty, color='yellow')

plt.xlim(0, 1280)

plt.ylim(720, 0)

## Quiz

In the below quiz, implement the following (see TO-DO's):

* Steps 2, 4 and 6 from above within the for loop in find\_lane\_pixels() - find the window boundaries, find all pixels within those boundaries, and if there are more than minpix, slide the window over to the mean of these pixels.
* Fit a polynomial to all the relevant pixels you've found in your sliding windows in fit\_polynomial().

# Skip the sliding windows step once you've found the lines

A picture containing clock

Description automatically generated

Great work! You've now built an algorithm that uses sliding windows to track the lane lines out into the distance. However, using the full algorithm from before and starting fresh on every frame may seem inefficient, as the lane lines don't necessarily move a lot from frame to frame.

In the next frame of video you don't need to do a blind search again, but instead you can just search in a margin around the previous lane line position, like in the above image. The green shaded area shows where we searched for the lines this time. So, once you know where the lines are in one frame of video, you can do a highly targeted search for them in the next frame.

This is equivalent to using a customized region of interest for each frame of video, and should help you track the lanes through sharp curves and tricky conditions. If you lose track of the lines, go back to your sliding windows search or other method to rediscover them.

Let's walk through one way to do this, and then you'll build it out further in a **quiz** below.

### Use the previous polynomial to skip the sliding window

In the previous quiz, we used left\_lane\_inds and right\_lane\_indsto hold the pixel values contained within the boundaries of a given sliding window. This time, we'll take the polynomial functions we fit before (left\_fit and right\_fit), along with a hyperparameter margin, to determine which activated pixels fall into the green shaded areas from the above image. Note that this margin can be a different value than the one originally used for your sliding windows!

To implement this in the below quiz, you'll want to grab only those pixels with x-values that are +/- your margin from your polynomial lines. Note that you'll only need to implement left\_lane\_inds and right\_lane\_inds in the quiz - most of the surrounding code, ignoring iterating through the windows, is the same as before!

The way we'll visualize this is a bit different than last time around, however, so make sure to pay attention to that if you want to visualize this step while working on your project.

## Quiz

In the below quiz, implement the following (see TO-DO's):

* Fit a polynomial to all the relevant pixels you've found in your sliding windows in fit\_poly().
* Set the area to search for activated pixels based on margin out from your fit polynomial within search\_around\_poly. Note that the quiz grader expects a margin of 100 pixels, but you can tune this as part of your own project!

## Fitting on Large Curves

One thing to consider in our current implementation of sliding window search is what happens when we arrive at the left or right edge of an image, such as when there is a large curve on the road ahead. If minpix is not achieved (i.e. the curve ran off the image), the starting position of our next window doesn't change, so it is just positioned directly above the previous window. This will repeat for however many windows are left in nwindows, stacking the sliding windows vertically against the side of the image, and likely leading to an imperfect polynomial fit.

Can you think of a way to solve this issue? If you want to tackle the curves on the harder challenge video as part of the project, you might want to include this in your lane finding algorithm.

# Measuring Curvature

You're getting very close to a final result! You have a thresholded image, where you've estimated which pixels belong to the left and right lane lines (shown in red and blue, respectively, below), and you've fit a polynomial to those pixel positions. Next we'll compute the radius of curvature of the fit.

A picture containing light, lit, clock, meter

Description automatically generated

Here I have fit the left and right lines with a second order polynomial shown in green.

In the last exercise, you located the lane line pixels, used their x and y pixel positions to fit a second order polynomial curve:

f(y) = Ay^2 + By + C*f*(*y*)=*Ay*2+*By*+*C*

You're fitting for f(y)*f*(*y*), rather than f(x)*f*(*x*), because the lane lines in the warped image are near vertical and may have the same x*x* value for more than one y*y* value.

### Radius of Curvature

The radius of curvature ([awesome tutorial here](http://www.intmath.com/applications-differentiation/8-radius-curvature.php)) at any point x*x* of the function x = f(y)*x*=*f*(*y*) is given as follows:

\LARGE R\_{curve} = \frac{[1 + (\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}*Rcurve*​=∣*dy*2*d*2*x*​∣[1+(*dydx*​)2]3/2​

In the case of the second order polynomial above, the first and second derivatives are:

\large f'(y) = \frac{dx}{dy} = 2Ay + B*f*′(*y*)=*dydx*​=2*Ay*+*B*

\large f''(y) = \frac{d^2x}{dy^2} = 2A*f*′′(*y*)=*dy*2*d*2*x*​=2*A*

So, our equation for radius of curvature becomes:

\LARGE R\_{curve} = \frac{(1 + (2Ay + B)^2)^{3/2}}{\left |2A \right |}*Rcurve*​=∣2*A*∣(1+(2*Ay*+*B*)2)3/2​

The y*y* values of your image increase from top to bottom, so if, for example, you wanted to measure the radius of curvature closest to your vehicle, you could evaluate the formula above at the y*y* value corresponding to the bottom of your image, or in Python, at yvalue = image.shape[0].

### Implementing the Calculation

#### Generate some fake data first

Most of the code below is just to generate some fake data to visualize with - everything up until the actual plotting of the data below should be replaced with your algorithms from before in your own implementation!

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

*# Generate some fake data to represent lane-line pixels*

ploty = np.linspace(0, 719, num=720)*# to cover same y-range as image*

quadratic\_coeff = 3e-4 *# arbitrary quadratic coefficient*

*# For each y position generate random x position within +/-50 pix*

*# of the line base position in each case (x=200 for left, and x=900 for right)*

leftx = np.array([200 + (y\*\*2)\*quadratic\_coeff + np.random.randint(-50, high=51)

**for** y **in** ploty])

rightx = np.array([900 + (y\*\*2)\*quadratic\_coeff + np.random.randint(-50, high=51)

**for** y **in** ploty])

leftx = leftx[::-1] *# Reverse to match top-to-bottom in y*

rightx = rightx[::-1] *# Reverse to match top-to-bottom in y*

*# Fit a second order polynomial to pixel positions in each fake lane line*

left\_fit = np.polyfit(ploty, leftx, 2)

left\_fitx = left\_fit[0]\*ploty\*\*2 + left\_fit[1]\*ploty + left\_fit[2]

right\_fit = np.polyfit(ploty, rightx, 2)

right\_fitx = right\_fit[0]\*ploty\*\*2 + right\_fit[1]\*ploty + right\_fit[2]

*# Plot up the fake data*

mark\_size = 3

plt.plot(leftx, ploty, 'o', color='red', markersize=mark\_size)

plt.plot(rightx, ploty, 'o', color='blue', markersize=mark\_size)

plt.xlim(0, 1280)

plt.ylim(0, 720)

plt.plot(left\_fitx, ploty, color='green', linewidth=3)

plt.plot(right\_fitx, ploty, color='green', linewidth=3)

plt.gca().invert\_yaxis() *# to visualize as we do the images*

The output looks like this:

A screenshot of a cell phone

Description automatically generated

#### Implementing the calculation itself

Now we have polynomial fits and we can calculate the radius of curvature.

In the below quiz, you'll implement the radius of curvature calculation (using our fake generated data from above - remember that you'll be using your own implemented algorithm in place of the generate\_data function in the below quiz!).

Use the \large R\_{curve}*Rcurve*​ equation above in order to calculate left\_curverad and right\_curverad in the measure\_curvature\_pixels() function below.

### From Pixels to Real-World

Great! You've now calculated the radius of curvature for our lane lines. But now we need to stop and think... We've calculated the radius of curvature based on pixel values, so the radius we are reporting is in pixel space, which is not the same as real world space. So we actually need to repeat this calculation after converting our x and y values to real world space.

This involves measuring how long and wide the section of lane is that we're projecting in our warped image. We could do this in detail by measuring out the physical lane in the field of view of the camera, but for this project, you can assume that if you're projecting a section of lane similar to the images above, the lane is about 30 meters long and 3.7 meters wide. Or, if you prefer to derive a conversion from pixel space to world space in your own images, compare your images with U.S. regulations that require a minimum lane width of 12 feet or 3.7 meters, and the dashed lane lines are 10 feet or 3 meters long each.

Let's say that our camera image has 720 relevant pixels in the y-dimension (remember, our image is perspective-transformed!), and we'll say roughly 700 relevant pixels in the x-dimension (our example of fake generated data above used from 200 pixels on the left to 900 on the right, or 700). Therefore, to convert from pixels to real-world meter measurements, we can use:

*# Define conversions in x and y from pixels space to meters*

ym\_per\_pix = 30/720 *# meters per pixel in y dimension*

xm\_per\_pix = 3.7/700 *# meters per pixel in x dimension*

In the below quiz, you'll use the above conversions in order to adjust your calculation from before to give real-world lane curvature values. Once again, you'll focus on the left\_curverad and right\_curverad values within the new measure\_curvature\_real() function; however, you'll also need to adjust how you use np.polyfit() within generate\_data() in order for this to work correctly. How do you need to change these to convert to meters?

* An insightful student has suggested an alternative approach which may scale more efficiently. That is, once the parabola coefficients are obtained, in pixels, convert them into meters. For example, if the parabola is x= a\*(y\*\*2) +b\*y+c; and mx and my are the scale for the x and y axis, respectively (in meters/pixel); then the scaled parabola is x= mx / (my \*\* 2) \*a\*(y\*\*2)+(mx/my)\*b\*y+c

Check out the [U.S. government specifications for highway curvature](http://onlinemanuals.txdot.gov/txdotmanuals/rdw/horizontal_alignment.htm#BGBHGEGC) to see how your numbers compare. There's no need to worry about absolute accuracy in this case, but your results should be "order of magnitude" correct.

## Additional Resources on Computer Vision

Nice work reaching the end of the computer vision content! While you still have the project left to do here, we're also providing some additional resources and recent research on the topic that you can come back to if you have time later on.

Reading research papers is a great way to get exposure to the latest and greatest in the field, as well as expand your learning. However, just like the project ahead, it's often best to learn by doing - if you find a paper that really excites you, try to implement it (or even something better) yourself!

##### Optional Reading

All of these are completely optional reading - you could spend hours reading through the entirety of these! We suggest moving onto the project first so you have what you’ve learned fresh on your mind, before coming back to check these out.

We've categorized these papers to hopefully help you narrow down which ones might be of interest, as well as highlighted a couple key reads by category by including their Abstract section, which summarizes the paper.

### Lane Finding with Semantic Segmentation

The below papers and resources concern a technique called semantic segmentation, where each pixel of an image gets classified individually!

[Fully Convolutional Networks for Semantic Segmentation](https://arxiv.org/abs/1605.06211) by E. Shelhamer, J. Long and T. Darrell

***Abstract:****Convolutional networks are powerful visual models that yield hierarchies of features. We show that convolutional networks by themselves, trained end-to-end, pixels-to-pixels, improve on the previous best result in semantic segmentation. Our key insight is to build "fully convolutional" networks that take input of arbitrary size and produce correspondingly-sized output with efficient inference and learning. We define and detail the space of fully convolutional networks, explain their application to spatially dense prediction tasks, and draw connections to prior models. We adapt contemporary classification networks (AlexNet, the VGG net, and GoogLeNet) into fully convolutional networks and transfer their learned representations by fine-tuning to the segmentation task. We then define a skip architecture that combines semantic information from a deep, coarse layer with appearance information from a shallow, fine layer to produce accurate and detailed segmentations. [...]*

You can use the [KITTI road dataset](http://www.cvlibs.net/datasets/kitti/eval_road.php) with the above technique for a model that can detect open space on the road.

[Lane Detection with Deep Learning (Part 1)](https://towardsdatascience.com/lane-detection-with-deep-learning-part-1-9e096f3320b7) and [(Part 2)](https://towardsdatascience.com/lane-detection-with-deep-learning-part-2-3ba559b5c5af) by M. Virgo

***Summary:****Udacity SDC student (and now Udacian!) investigates using a deep learning approach to lane detection in order to improve upon the Advanced Lane Finding project, eventually building a model with a fully convolutional neural network that detects the road is a wider variety of situations and at faster speed.*

### Other Lane Finding Techniques

The below paper uses a multi-task model to identify lane and road markings, as well as vanishing point of the road, in order to build a robust model.

[VPGNet: Vanishing Point Guided Network for Lane and Road Marking Detection and Recognition](https://arxiv.org/abs/1710.06288) by S. Lee, et. al.

***Abstract:****In this paper, we propose a unified end-to-end trainable multi-task network that jointly handles lane and road marking detection and recognition that is guided by a vanishing point under adverse weather conditions. We tackle rainy and low illumination conditions [...] At night, color distortion occurs under limited illumination. As a result, no benchmark dataset exists and only a few developed algorithms work under poor weather conditions. To address this shortcoming, we build up a lane and road marking benchmark which consists of about 20,000 images with 17 lane and road marking classes under four different scenarios: no rain, rain, heavy rain, and night. We train and evaluate several versions of the proposed multi-task network and validate the importance of each task. The resulting approach, VPGNet, can detect and classify lanes and road markings, and predict a vanishing point with a single forward pass. Experimental results show that our approach achieves high accuracy and robustness under various conditions in real-time (20 fps). [...]*

### Vehicle Detection

The below paper builds a model to both detect vehicles as well as estimate their dimensions along the road.

[Learning to Map Vehicles into Bird's Eye View](https://arxiv.org/abs/1706.08442) by A. Palazzi, et. al.

***Abstract:****Awareness of the road scene is an essential component for both autonomous vehicles and Advances Driver Assistance Systems and is gaining importance both for the academia and car companies. This paper presents a way to learn a semantic-aware transformation which maps detections from a dashboard camera view onto a broader bird's eye occupancy map of the scene. To this end, a huge synthetic dataset featuring 1M couples of frames, taken from both car dashboard and bird's eye view, has been collected and automatically annotated. A deep-network is then trained to warp detections from the first to the second view. We demonstrate the effectiveness of our model against several baselines and observe that is able to generalize on real-world data despite having been trained solely on synthetic ones.*

You may have noticed a lot of the papers above include deep learning techniques, which are now commonly used in many computer vision applications. More on deep learning is coming up!