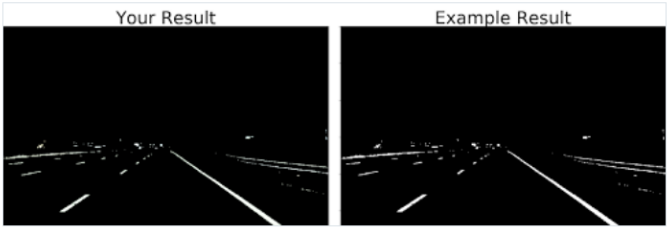
Color selection

```python
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

# Read in the image
image = mpimg.imread('test.jpg')

# Grab the x and y size and make a copy of the image
ysize = image.shape[0]
xsize = image.shape[1]
color_select = np.copy(image)

# Define color selection criteria
###### MODIFY THESE VARIABLES TO MAKE YOUR COLOR SELECTION
red_threshold = 200
green_threshold = 200
blue_threshold = 200
######

rgb_threshold = [red_threshold, green_threshold, blue_threshold]

# Do a boolean or with the "|" character to identify
# pixels below the thresholds
thresholds = (image[:,:,0] < rgb_threshold[0]) \
            | (image[:,:,1] < rgb_threshold[1]) \
            | (image[:,:,2] < rgb_threshold[2])
color_select[thresholds] = [0,0,0]

# Display the image
plt.imshow(color_select)
```



Canny edge detection

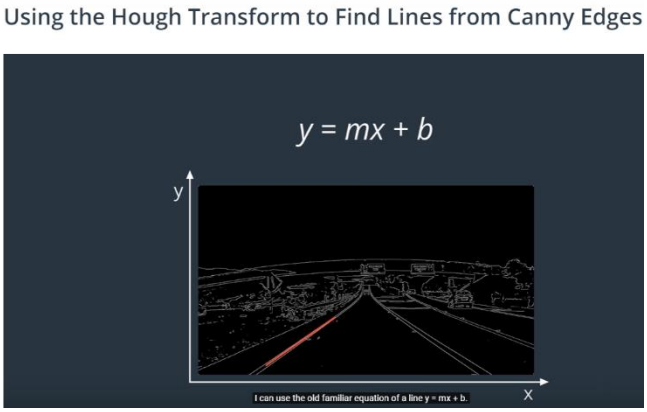
It is an improvement of the previous version. Color and light conditions do not affect.

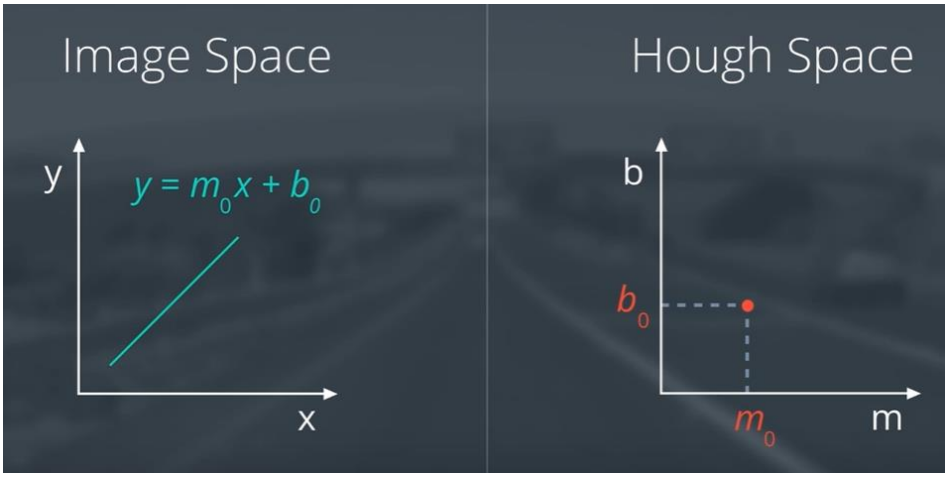
The algorithm maps the grey version of the original image into a one black and white (representing edges) by taking the highest gradients in the image. Then it shrinks the output into thin lines.

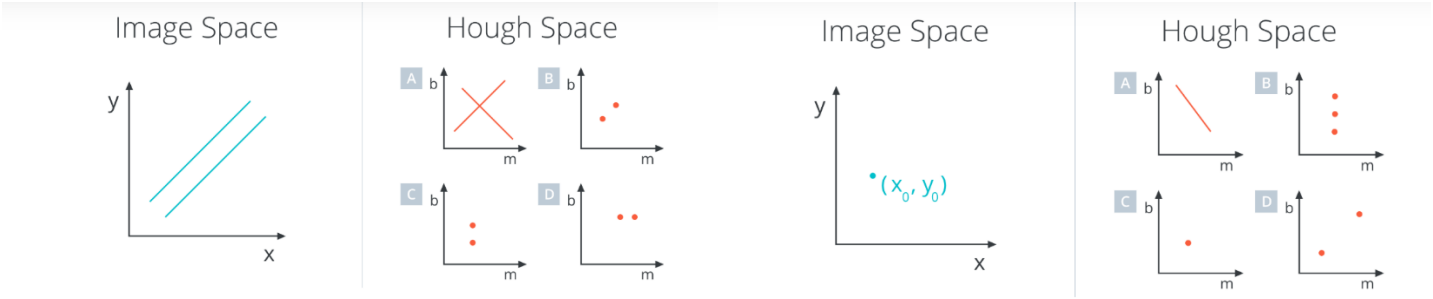
```
edges = cv2.Canny(gray,
low_threshold, high_threshold)
```

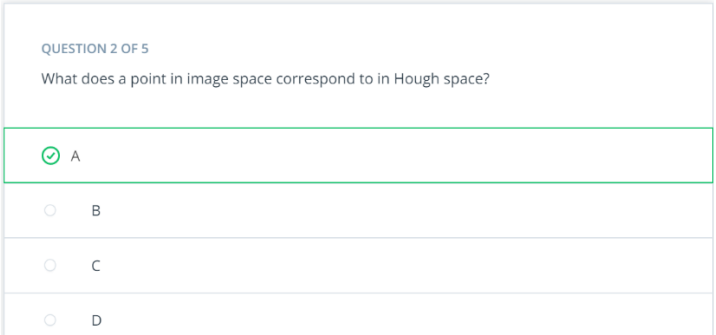
```python
# Do all the relevant imports
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2

# Read in the image and convert to grayscale
# Note: in the previous example we were reading a .jpg
# Here we read a .png and convert to 0,255 bytescale
image = mpimg.imread('exit-ramp.jpg')
gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)

# Define a kernel size for Gaussian smoothing / blurring
kernel_size = 5 # Must be an odd number (3, 5, 7...)
blur_gray = cv2.GaussianBlur(gray,(kernel_size, kernel_size),0)

# Define our parameters for Canny and run it
low_threshold = 50
high_threshold = low_threshold * 3 # between 2:1 and 3:1 recommended
edges = cv2.Canny(blur_gray, low_threshold, high_threshold)

# Display the image
plt.imshow(edges, cmap='Greys_r')
```

Hough Transformation



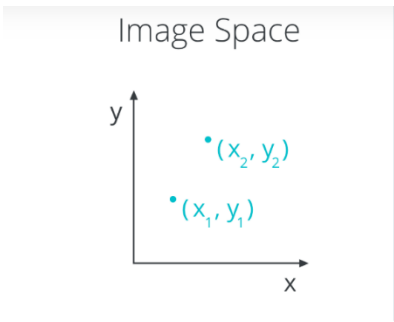


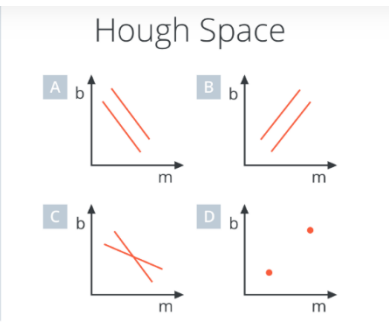


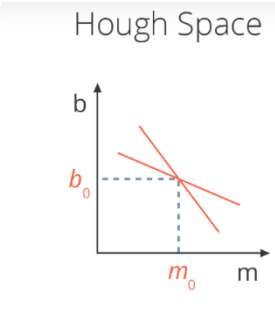


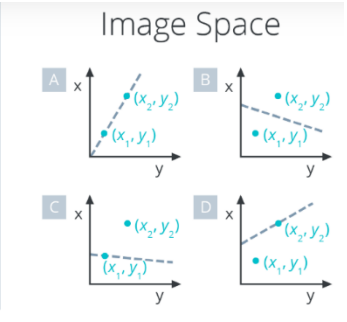
A point in image space describes a line in Hough space. So a line in an image is a point in Hough space and a point in an image is a line in Hough space… cool!

Rearranging the equation of a line, we find that a single point (x,y) corresponds to the line b = y - xm









QUESTION 3 OF 5

What is the representation in Hough space of two points in image space?

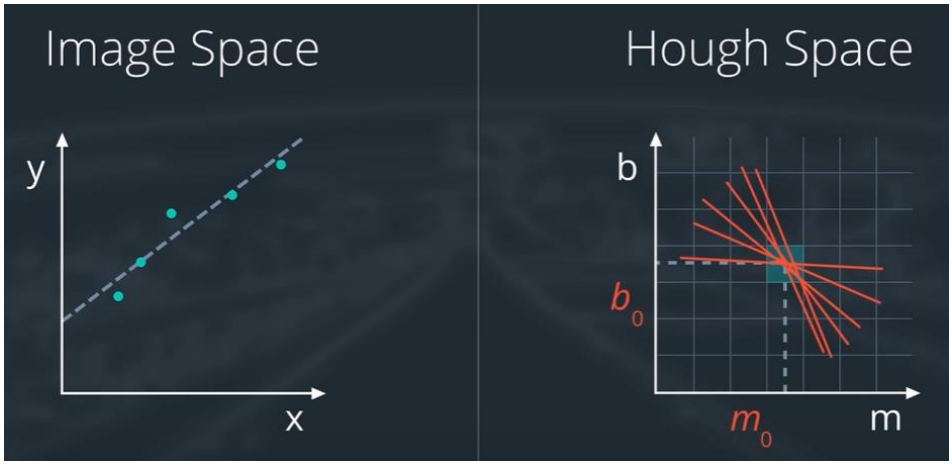
- A
- B
- ✓ C
- D

QUESTION 4 OF 5

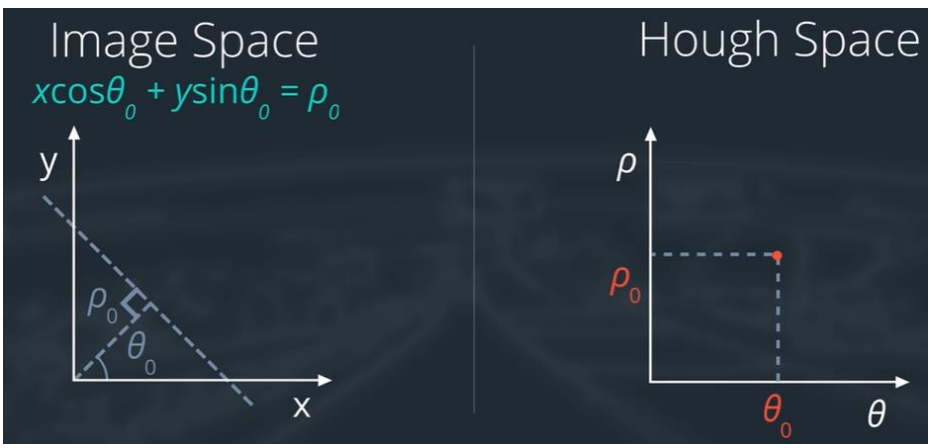
What does the intersection point of the two lines in Hough space correspond to in image space?

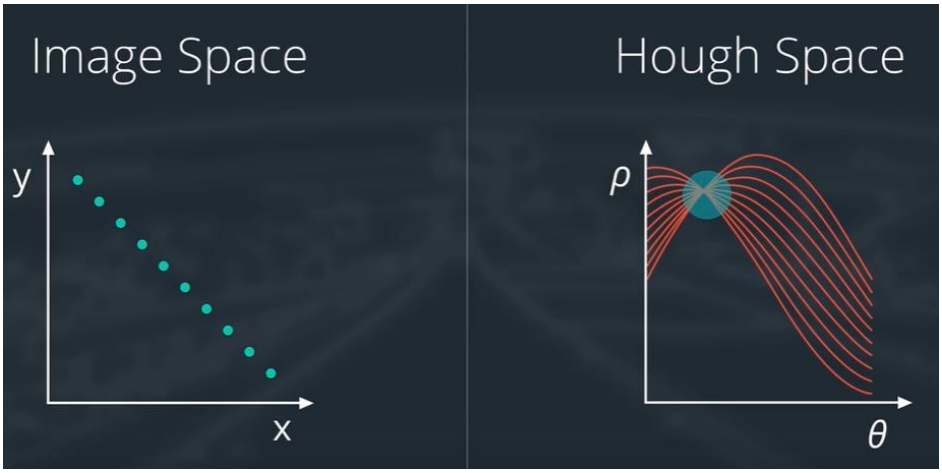
- ✓ A) A line in image space that passes through both (x1, y1) and (x2, y2)
- B) A line in image space that passes between (x1, y1) and (x2, y2)
- C) A line in image space that passes through (x1, y1)
- D) A line in image space that passes through only (x2, y2)

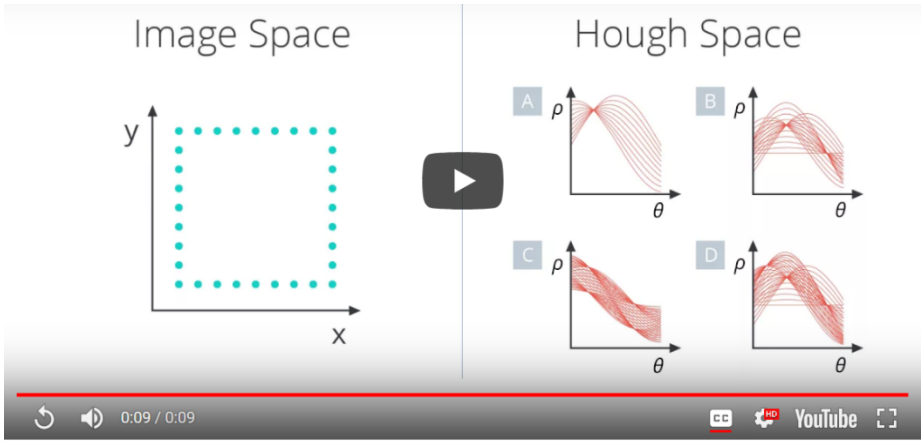
The intersection point at (m_0, b_0) represents the line $y = m_0x + b_0$ in image space and it must be the line that passes through both points!



The problem is vertical lines have infinite slope. So we transform to polar coordinates







QUESTION 5 OF 5

What happens if we run a Hough Transform on an image of a square? What will the corresponding plot in Hough space look like?

- A
- B
- C ✓
- D

The four major intersections between curves in Hough space correspond to the four sides of the square.

Let's look at the input parameters for the OpenCV function `HoughLinesP` that we will use to find lines in the image. You will call it like this:

```
lines = cv2.HoughLinesP(masked_edges, rho, theta, threshold, np.array([]),
                                            min_line_length, max_line_gap)
```

In this case, we are operating on the image `masked_edges` (the output from `Canny`) and the output from `HoughLinesP` will be `lines`, which will simply be an array containing the endpoints (x_1, y_1, x_2, y_2) of all line segments detected by the transform operation. The other parameters define just what kind of line segments we're looking for.

First off, `rho` and `theta` are the distance and angular resolution of our grid in Hough space. Remember that, in Hough space, we have a grid laid out along the (Θ, ρ) axis. You need to specify `rho` in units of pixels and `theta` in units of radians.

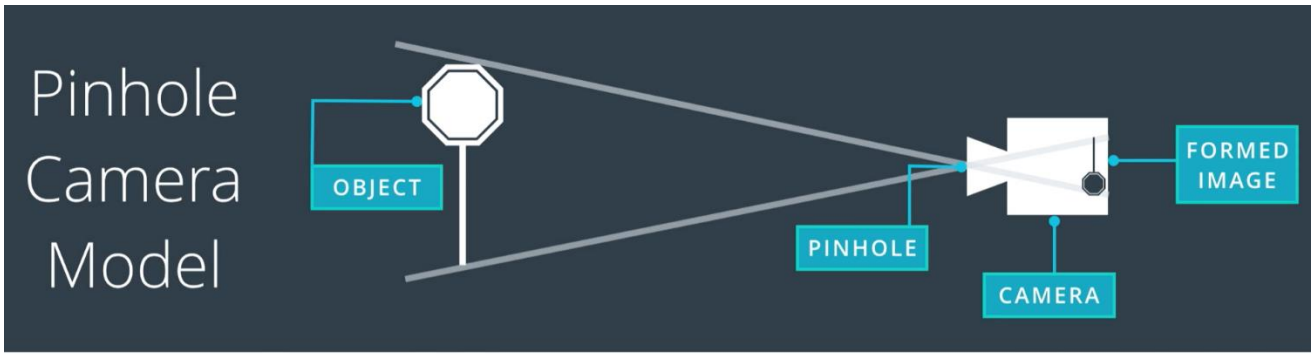
So, what are reasonable values? Well, rho takes a minimum value of 1, and a reasonable starting place for theta is 1 degree (pi/180 in radians). Scale these values up to be more flexible in your definition of what constitutes a line.

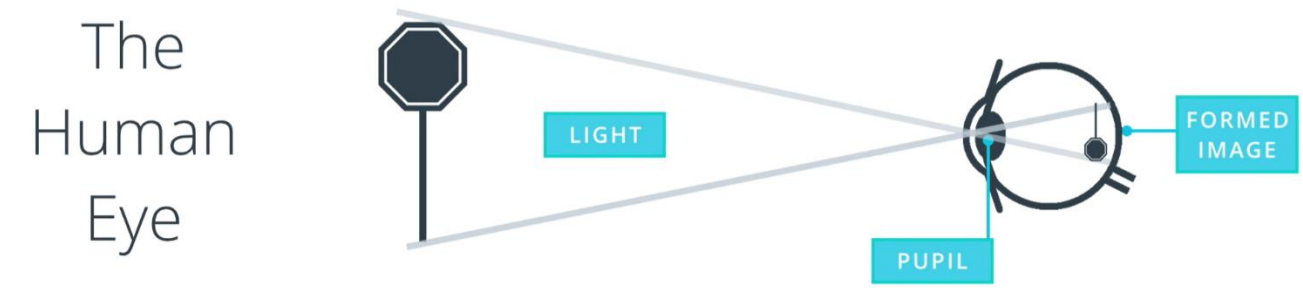
The `threshold` parameter specifies the minimum number of votes (intersections in a given grid cell) a candidate line needs to have to make it into the output. The empty `np.array([])` is just a placeholder, no need to change it. `min_line_length` is the minimum length of a line (in pixels) that you will accept in the output, and `max_line_gap` is the maximum distance (again, in pixels) between segments that you will allow to be connected into a single line. You can then iterate through your output `lines` and draw them onto the image to see what you got!

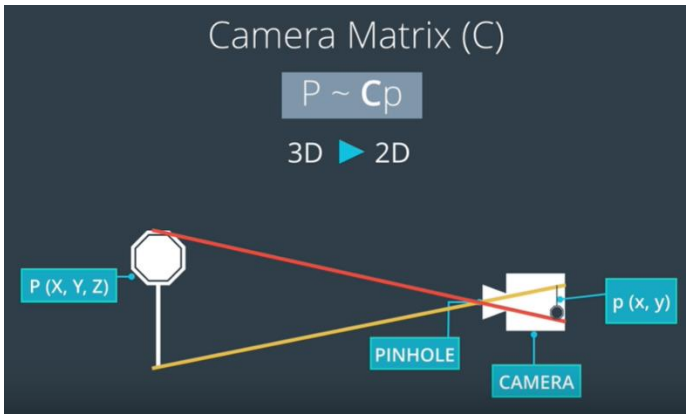
QUIZ QUESTION

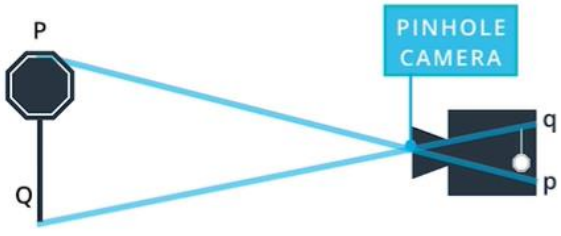
Why is it important to correct for image distortion?

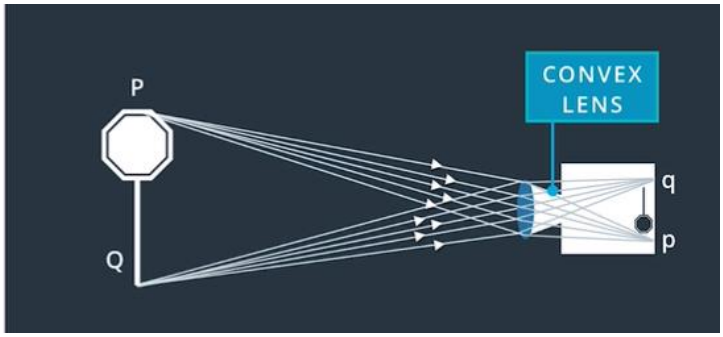
- Distortion can change the apparent size of an object in an image.
- Distortion can change the apparent shape of an object in an image.
- Distortion can cause an object's appearance to change depending on where it is in field of view.
- Distortion can make objects appear closer or farther away than they actually are.

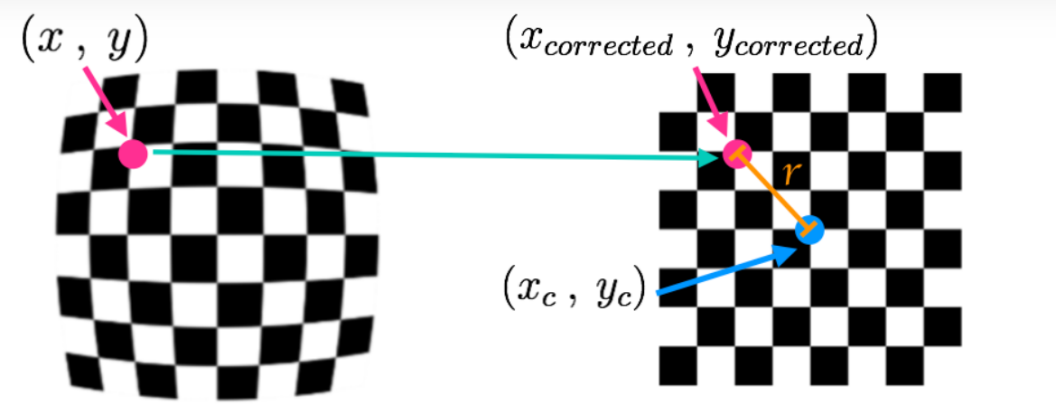












Points in a distorted and undistorted (corrected) image. The point (x, y) is a single point in a distorted image and (x_corrected, y_corrected) is where that point will appear in the undistorted (corrected) image.

$$x_{distorted} = x_{ideal}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{distorted} = y_{ideal}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Radial distortion correction.

There are two more coefficients that account for **tangential distortion**: **p1** and **p2**, and this distortion can be corrected using a different correction formula.

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$

$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

Tangential distortion correction.

QUIZ QUESTION

Fun fact: the "pinhole camera" is not just a model, but was actually the earliest means of projecting images, first documented almost 2500 years ago in China!

Now back to computer vision... what is the fundamental difference between images formed with a pinhole camera and those formed using lenses?

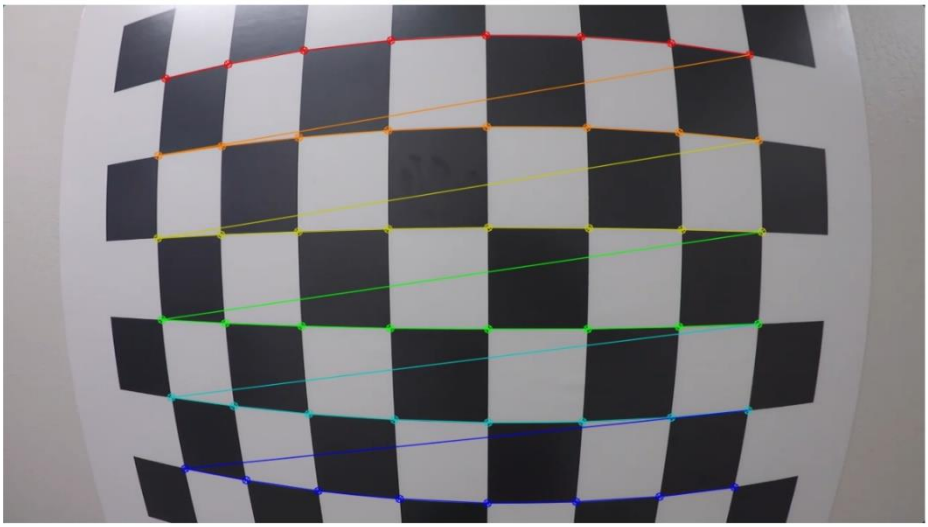
- Images formed by a pinhole camera are always flipped upside down, lensed images appear upright.
- Pinhole camera images are black and white, while lenses form color images.
- ✓ Pinhole camera images are free from distortion, but lenses tend to introduce image distortion.
- Images formed by pinhole cameras are the same as those formed using lenses.

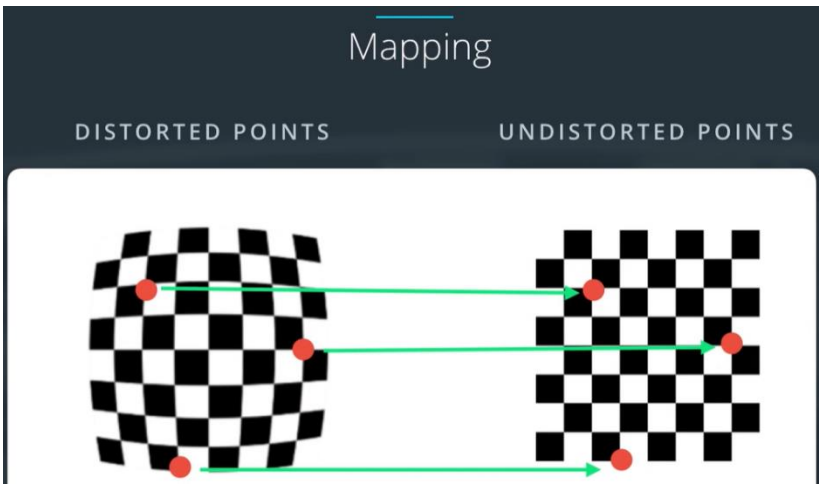
Finding Corners

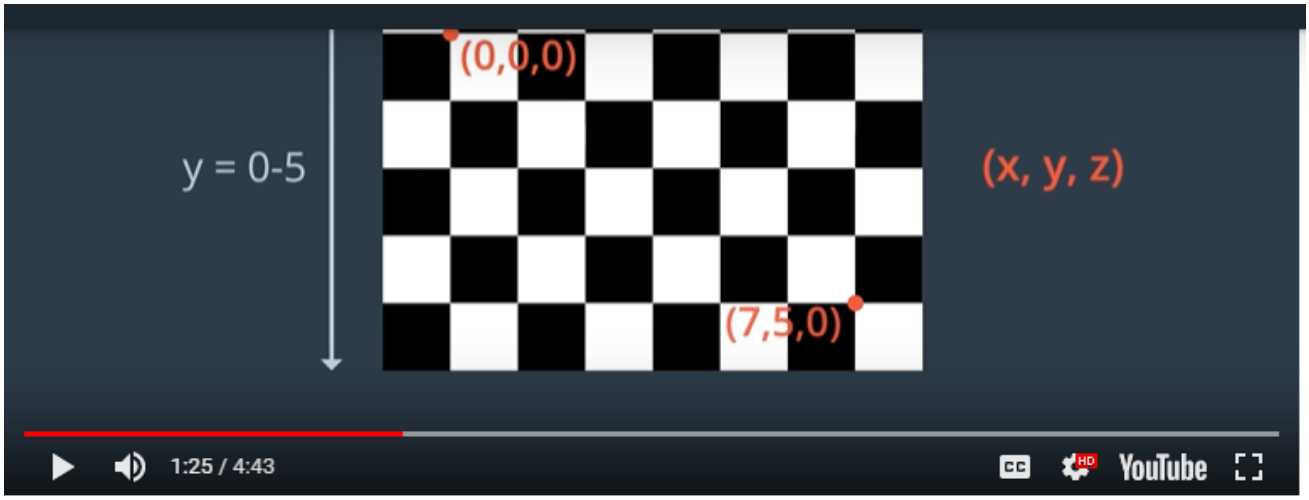
In this exercise, you'll use the OpenCV functions `findChessboardCorners()` and `drawChessboardCorners()` to automatically find and draw corners in an image of a chessboard pattern.

To learn more about both of those functions, you can have a look at the OpenCV documentation here: cv2.findChessboardCorners() and cv2.drawChessboardCorners().

Applying these two functions to a sample image, you'll get a result like this:







Note Regarding Corner Coordinates

Since the origin corner is (0,0,0) the final corner is (6,4,0) relative to this corner rather than (7,5,0).

Examples of Useful Code

Converting an image, imported by cv2 or the glob API, to grayscale:

```
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```

Note: If you are reading in an image using mpimg.imread() this will read in an **RGB** image and you should convert to grayscale using *cv2.COLOR_RGB2GRAY*, but if you are using cv2.imread() or the glob API, as happens in this video example, this will read in a **BGR** image and you should convert to grayscale using *cv2.COLOR_BGR2GRAY*. We'll learn more about color conversions later on in this lesson, but please keep this in mind as you write your own code and look at code examples.

Finding chessboard corners (for an 8x6 board):

```
ret, corners = cv2.findChessboardCorners(gray, (8,6), None)
```

Drawing detected corners on an image:

```
img = cv2.drawChessboardCorners(img, (8,6), corners, ret)
```

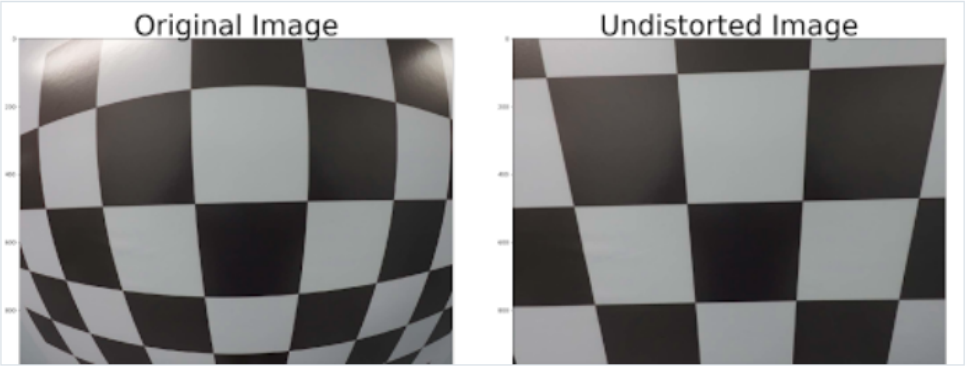
Camera calibration, given object points, image points, and the **shape of the grayscale image**:

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::
```

Undistorting a test image:

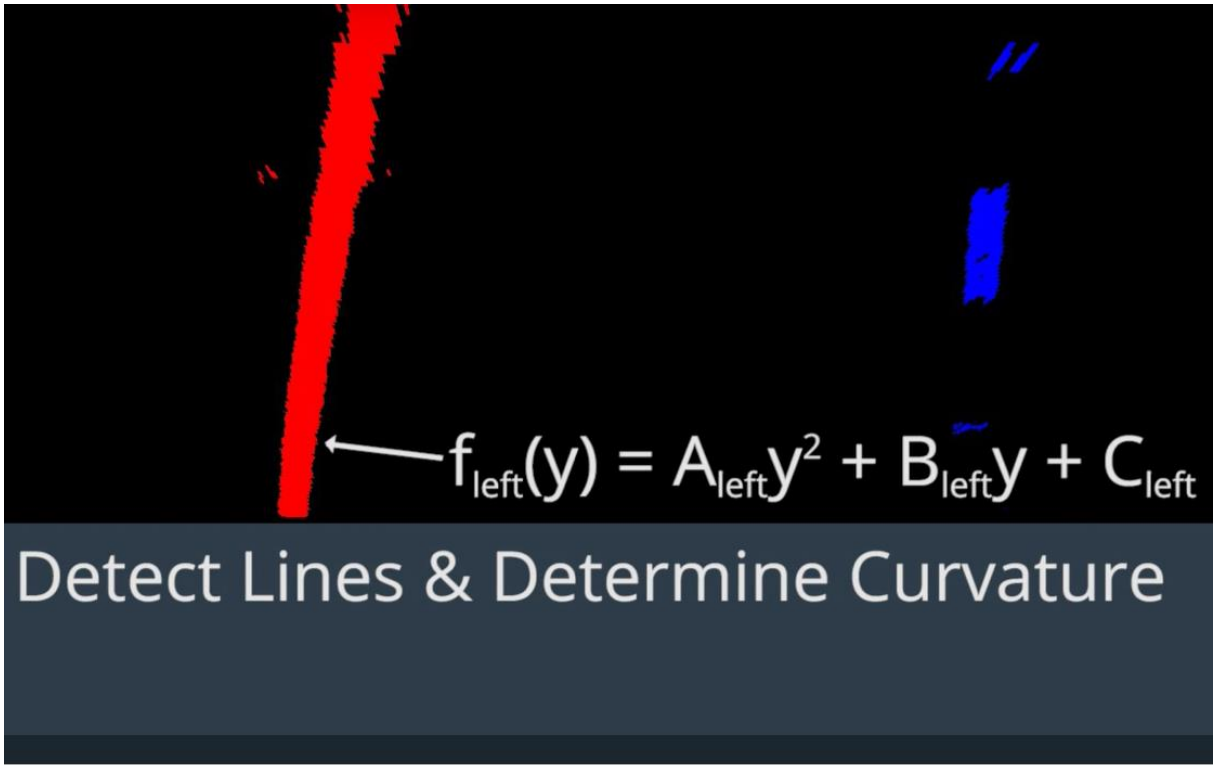
```
dst = cv2.undistort(img, mtx, dist, None, mtx)
```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Read in the saved objpoints and imgpoints
dist_pickle = pickle.load( open( "wide_dist_pickle.p", "rb" ) )
objpoints = dist_pickle["objpoints"]
imgpoints = dist_pickle["imgpoints"]

# Read in an image
img = cv2.imread('test_image.png')

# TODO: Write a function that takes an image, object points, and image points
# performs the camera calibration, image distortion correction and
# returns the undistorted image
def cal_undistort(img, objpoints, imgpoints):
    # Use cv2.calibrateCamera() and cv2.undistort()
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img.shape[1:]
    undist = cv2.undistort(img, mtx, dist, None, mtx)
    return undist

undistorted = cal_undistort(img, objpoints, imgpoints)

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
f.tight_layout()
ax1.imshow(img)
ax1.set_title('Original Image', fontsize=50)
ax2.imshow(undistorted)
ax2.set_title('Undistorted Image', fontsize=50)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```



Lane curvature is detected through this pipeline:

Image → Thresholding to get the lanes → Masking ROI → Perspective Transformation → Detect Lines and detect curvature



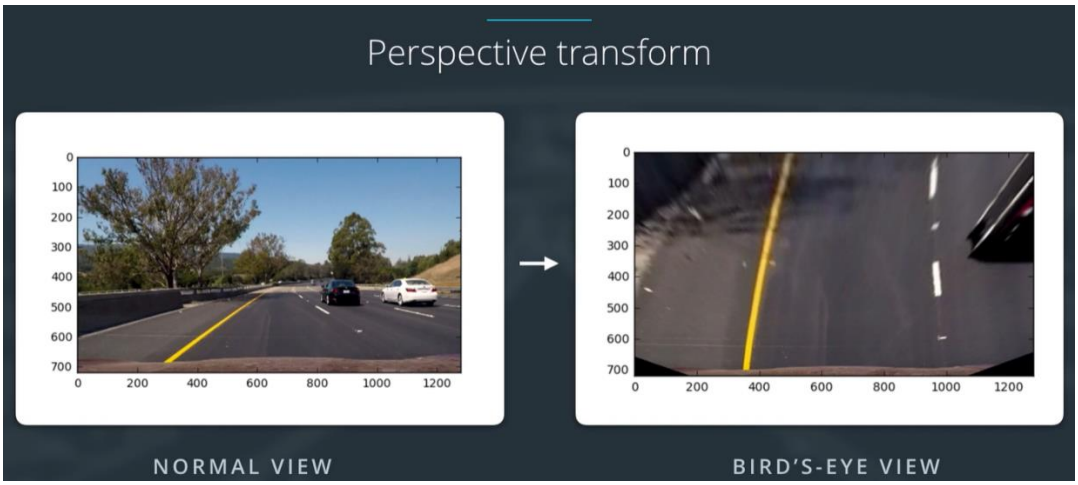
Calculating Lane Curvature

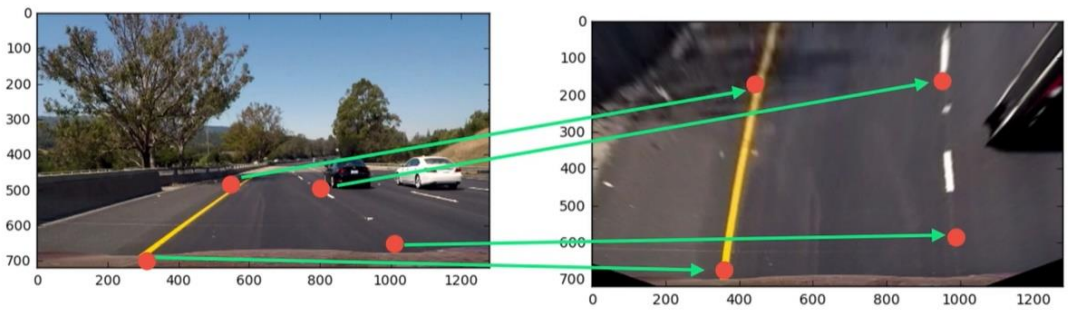
Self-driving cars need to be told the correct steering angle to turn, left or right. You can calculate this angle if you know a few things about the speed and dynamics of the car and how much the lane is curving.

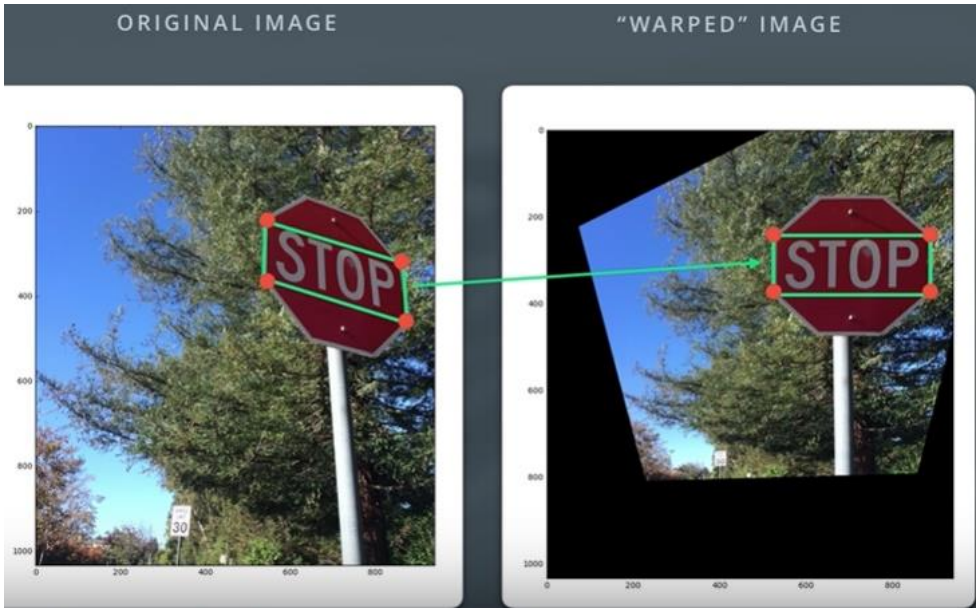
One way to calculate the curvature of a lane line, is to fit a 2nd degree polynomial to that line, and from this you can easily extract useful information.

For a lane line that is close to vertical, you can fit a line using this formula: **f(y) = Ay^2 + By + C**, where A, B, and C are coefficients.

A gives you the curvature of the lane line, B gives you the heading or direction that the line is pointing, and C gives you the position of the line based on how far away it is from the very left of an image (y = 0)







PROCESS	PURPOSE
Camera Calibration	To compute the transformation between 3D object points in the world and 2D image points.
Distortion Correction	To ensure that the geometrical shape of objects is represented consistently, no matter where they appear in an image.
Perspective Transform	To transform an image such that we are effectively viewing objects from a different angle or direction.

Sobel Operator

The Sobel operator is at the heart of the Canny edge detection algorithm you used in the Introductory Lesson. Applying the Sobel operator to an image is a way of taking the derivative of the image in the x or y direction. The operators for $Sobel_x$ and $Sobel_y$, respectively, look like this:

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

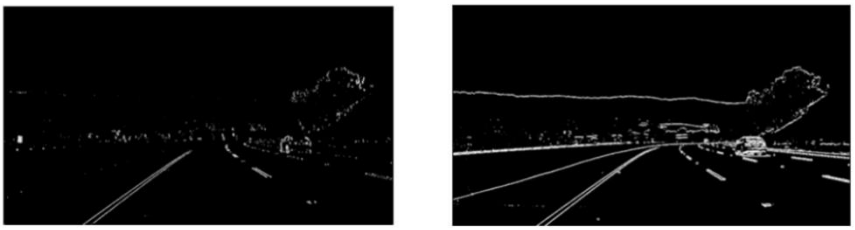
$$S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

These are examples of Sobel operators with a kernel size of 3 (implying a 3 x 3 operator in each case). This is the minimum size, but the kernel size can be any odd number. A larger kernel implies taking the gradient over a larger region of the image, or, in other words, a smoother gradient.

To understand how these operators take the derivative, you can think of overlaying either one on a 3 x 3 region of an image. If the image is flat across that region (i.e., there is little change in values across the given region), then the result (summing the element-wise product of the operator and corresponding image pixels) will be zero.

$$gradient = \sum(region * S_x)$$

And then we take the absolute value, we get the result:



Sobel x Sobel y

Examples of Useful Code

You need to pass a single color channel to the `cv2.Sobel()` function, so first convert it to grayscale:

```
gray = cv2.cvtColor(im, cv2.COLOR_RGB2GRAY)
```

Note: Make sure you use the correct grayscale conversion depending on how you've read in your images. Use `cv2.COLOR_RGB2GRAY` if you've read in an image using `mpimg.imread()`. Use `cv2.COLOR_BGR2GRAY` if you've read in an image using `cv2.imread()`.

Calculate the derivative in the x direction (the 1, 0 at the end denotes x direction):

```
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
```

Calculate the derivative in the y direction (the 0, 1 at the end denotes y direction):

```
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1)
```

Calculate the absolute value of the x derivative:

```
abs_sobelx = np.absolute(sobelx)
```

Convert the absolute value image to 8-bit:

```
scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))
```

Note: It's not entirely necessary to convert to 8-bit (range from 0 to 255) but in practice, it can be useful in the event that you've written a function to apply a particular threshold, and you want it to work the same on input images of different scales, like jpg vs. png. You could just as well choose a different standard range of values, like 0 to 1 etc.

Create a binary threshold to select pixels based on gradient strength:

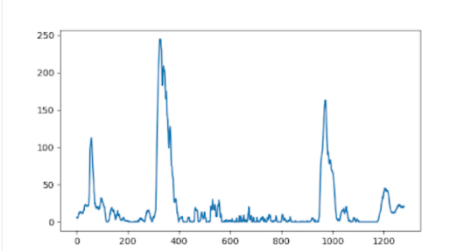
```
thresh_min = 20
thresh_max = 100
sxbinary = np.zeros_like(scaled_sobel)
sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1
plt.imshow(sxbinary, cmap='gray')
```

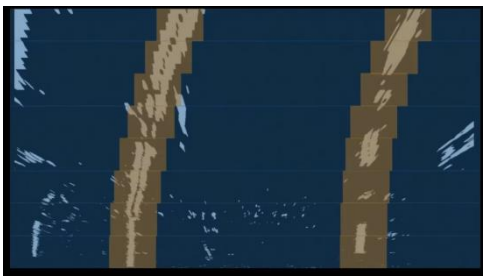
Result

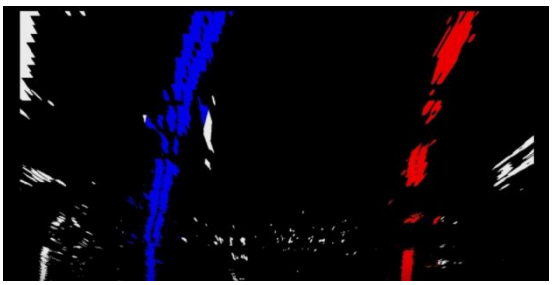
Pixels have a value of 1 or 0 based on the strength of the x gradient.

lane_histogram.py solution.py

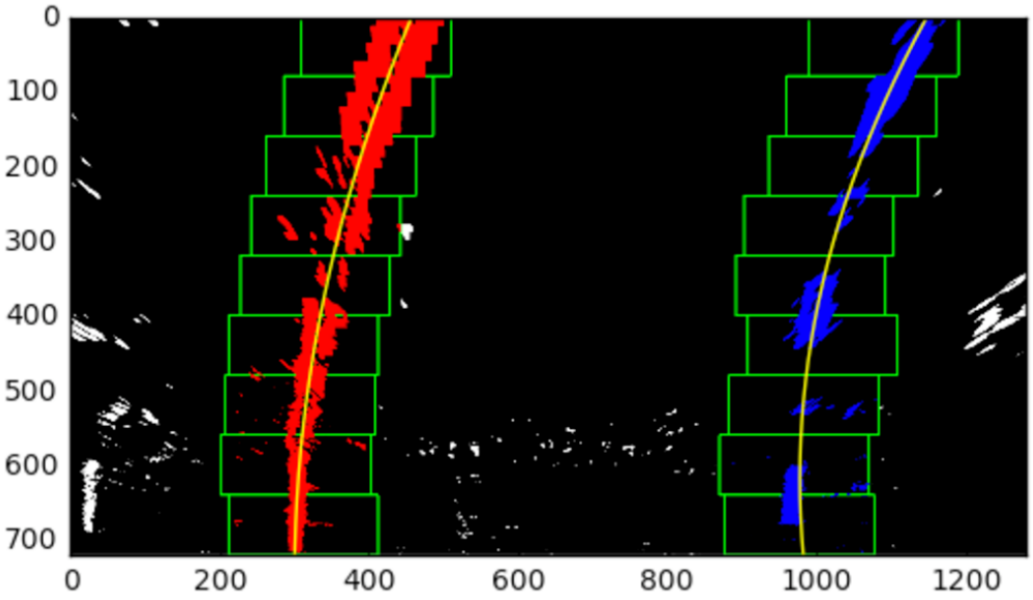
```
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Load our image
# `mpimg.imread` will load .jpg as 0-255, so normalize back to 0-1
img = mpimg.imread('warped_example.jpg')/255

def hist(img):
    # TO-DO: Grab only the bottom half of the image
    # Lane lines are likely to be mostly vertical nearest to the car
    bottom_half = img[img.shape[0] // 2 :, :]

    # TO-DO: Sum across image pixels vertically - make sure to set `axis`
    # i.e. the highest areas of vertical lines should be larger values
    histogram = np.sum(bottom_half, axis = 0)

    return histogram

# Create histogram of image binary activations
histogram = hist(img)

# Visualize the resulting histogram
plt.plot(histogram)
```







Implement Sliding Windows and Fit a Polynomial



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]
```

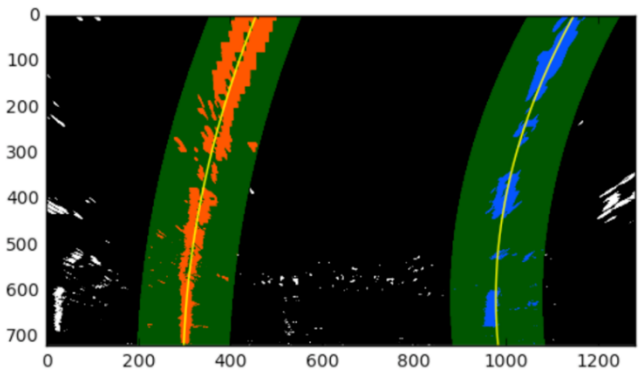
Take note of *how* we fit the lines above - while normally you calculate a y-value for a given x, here we do the opposite. Why? Because we expect our lane lines to be (mostly) vertically-oriented.

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)
```

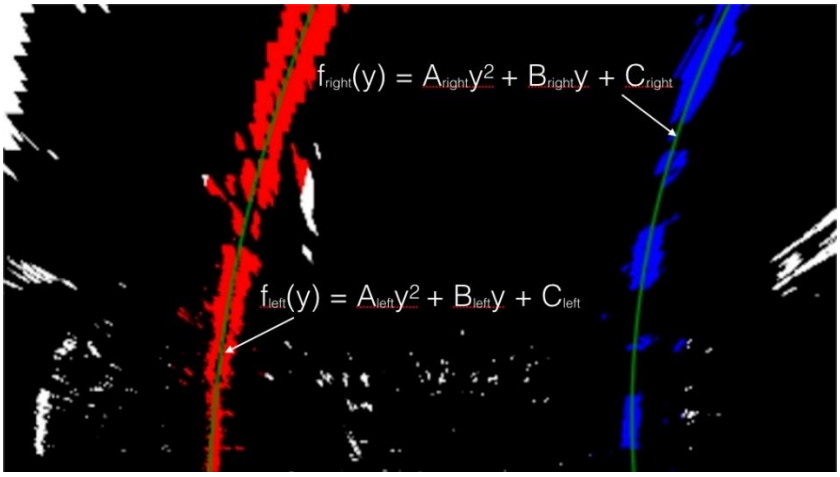
Skip the sliding windows step once you've found the lines



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.



Radius of Curvature

The radius of curvature (awesome tutorial here) at any point x of the function $x = f(y)$ is given as follows:

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

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

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

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

So, our equation for radius of curvature becomes:

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

The 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 value corresponding to the bottom of your image, or in Python, at `yvalue = image.shape[0]`.

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
```