

# Fourier Transform

## Goal

In this section, we will learn

- To find the Fourier Transform of images using OpenCV
- To utilize the FFT functions available in Numpy
- Some applications of Fourier Transform
- We will see following functions : `cv2.dft()`, `cv2.idft()` etc

## Theory

Fourier Transform is used to analyze the frequency characteristics of various filters. For images, **2D Discrete Fourier Transform (DFT)** is used to find the frequency domain. A fast algorithm called **Fast Fourier Transform (FFT)** is used for calculation of DFT. Details about these can be found in any image processing or signal processing textbooks. Please see [Additional Resources](#) section.

For a sinusoidal signal,  $x(t) = A \sin(2\pi ft)$ , we can say  $f$  is the frequency of signal, and if its frequency domain is taken, we can see a spike at  $f$ . If signal is sampled to form a discrete signal, we get the same frequency domain, but is periodic in the range  $[-\pi, \pi]$  or  $[0, 2\pi]$  (or  $[0, N]$  for N-point DFT). You can consider an image as a signal which is sampled in two directions. So taking Fourier transform in both X and Y directions gives you the frequency representation of image.

More intuitively, for the sinusoidal signal, if the amplitude varies so fast in short time, you can say it is a high frequency signal. If it varies slowly, it is a low frequency signal. You can extend the same idea to images. Where does the amplitude varies drastically in images? At the edge points, or noises. So we can say, edges and noises are high frequency contents in an image. If there is no much changes in amplitude, it is a low frequency component. ( Some links are added to [Additional Resources](#) which explains frequency transform intuitively with examples).

Now we will see how to find the Fourier Transform.

Edge and Noises are high frequency contents in an image

## Fourier Transform in Numpy

First argument => is the input image which is grayscale

Second argument => is the optional decides the size of output array

If (size\_image < input argument ) => is padded

If (size\_image > input\_argument ) => it will be cropped

Iff (no\_argument\_passed ) => output array size will be same as input

First we will see how to find Fourier Transform using Numpy. Numpy has an FFT package to do this. `np.fft.fft2()` provides us the **frequency transform** which will be a **complex array**. Its first argument is the **input image**, which is **grayscale**. Second argument is optional which decides the **size of output array**. If it is greater than size of input image, input image is padded **with zeros before calculation of FFT**. If it is less than input image, **input image will be cropped**. If no arguments passed, Output array size will be same as input.

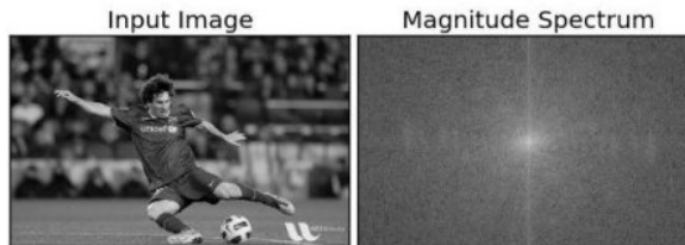
Now once you got the result, **zero frequency component (DC component)** will be at **top left** corner. If you want to bring it to center, you need to shift the result by  $\frac{N}{2}$  in both the directions. This is simply done by the function, `np.fft.fftshift()`. (It is more easier to analyze). Once you found the frequency transform, you can find the **magnitude spectrum**.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg',0)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
magnitude_spectrum = 20*np.log(np.abs(fshift))

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([], plt.yticks([]))
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([], plt.yticks([]))
plt.show()
```

Result look like below:



```
import cv2

import numpy as np

from matplotlib import pyplot as plt

img = cv2.imread('H:\\Udacity - Computer Vision Nanodegree v1.0.0\\Part 01-Module 01-Lesson 03_Convolutional Filters and Edge Detection\\Search\\image_test.jpg',0)

f = np.fft.fft2(img) # to fouier_transform

fshift = np.fft.fftshift(f) # to shift in center

magnitude_spectrum = 20*np.log(np.abs(fshift))

plt.subplot(121),plt.imshow(img, cmap = 'gray')

plt.title('Input Image'), plt.xticks([], plt.yticks([]))

plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')

plt.title('Magnitude Spectrum'), plt.xticks([], plt.yticks([]))

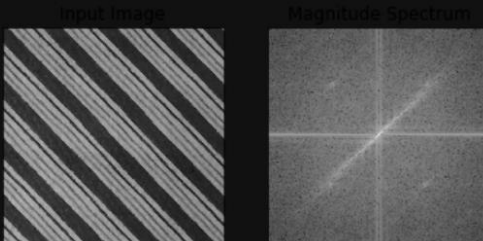
plt.show()
```

## Quiz

```
[5]: import cv2
import numpy as np
from matplotlib import pyplot as plt

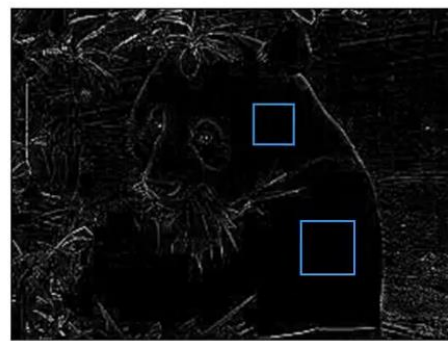
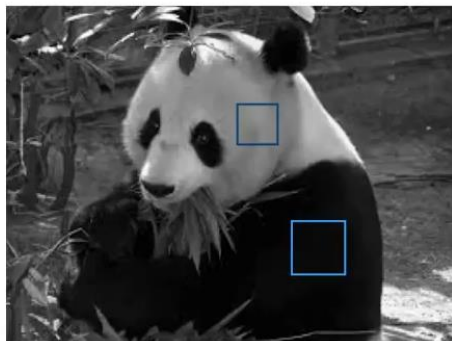
img = cv2.imread('H:\\Udacity - Computer Vision Nanodegree v1.0.0\\Part 01-Module 01-Lesson 03_Convolutional
Filters and Edge Detection\\Search\\fourier_transform_quiz.png',0)
f = np.fft.fft2(img) # to fouier_transform
fshift = np.fft.fftshift(f) # to shift in center
magnitude_spectrum = 20*np.log(np.abs(fshift))

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()
```



High pass filter => are used to make image appear sharper and enhance high frequency parts of image which are areas where the level of intensity in neighbor pixels rapidly change like from very dark to very light pixel

## HIGH-PASS FILTERS



a high-pass filter will black these areas out and turn the pixels black,

## Convolution network

A kernel is a matrix of numbers that modifies an image

High pass filter it is edge detection filter

It's a three by three kernel whose elements all sum to zero .

0	-1	0
-1	4	-1
0	-1	0

edge detection filter

The sum of element is equal = Zero , this filter is computing the difference or change between **neighboring pixels**

Differences are calculated by subtracting pixel values from one another

If this kernel values did not add up to zeros it is positively or negatively weighted which will have the effect of brightening or darkening the entire filtered image respectively

This is called I will call k , this is called kernel convolutional and convolution is represented by an

K

## CONVOLUTION

0	-1	0
-1	4	-1
0	-1	0

\*



K

$F(x,y)$

$K * F(x,y)$  = output image it is represented for asterisk not to be mistaken for a multiplication

Kernel operation is important operation in computer vision applications and it's the basis for conventutional neural networks

To better see the pixel operation

## CONVOLUTION

0	-1	0
-1	4	-1
0	-1	0

0	5	12	16	
2	10	18	20	
5	20	45	75	100
	50	80	105	120
		100	110	170
			225	220
				255

# CONVOLUTION

0	-1	0
-1	4	-1
0	-1	0

0	5	12	16		
2	10	18	20		
5	20	45	75	100	
	50	80	105	120	140
	100	110	170	225	220
				255	250

0	-140	0
-225	880	-205
0	-250	0

**= 60**

This value means a very small edge has been detected,

Which are negative weights that increase the contrast in the image .

The corners are the farthest away from the center pixel and in this example

Sobel filter => it edge detection

- 1- Vertical
- 2- Horizontal

Vertical

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} * A$$

and

Horizontal

$$G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * A$$



## Sobel Filter example

Convolve the Sobel kernels to the original image

10	50	10	50	10
10	55	10	55	10
10	65	10	65	10
10	50	10	50	10
10	55	10	55	10

Original image

1	0	-1
2	0	-2
1	0	-1

-1	-2	-1
0	0	0
1	2	1



## Sobel filter example

- Compute  $G_x$  and  $G_y$ , gradients of the image performing the convolution of Sobel kernels with the image
- Use zero-padding to extend the image

0	0	10	10	10
0	0	10	10	10
0	0	10	10	10
0	0	10	10	10
0	0	10	10	10

y

x

1	0	-1
2	0	-2
1	0	-1

$h_x$

-1	-2	-1
0	0	0
1	2	1

$h_y$

$G_x$

0	30	30	0	-30
0	40	40	0	-40
0	40	40	0	-40
0	40	40	0	-40
0	30	30	0	-30

$G_y$

-10	-30	-40	-30	-10
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
10	30	40	30	10

# Gradients

Gradients are a measure of intensity change in an image, and they generally mark object boundaries and changing area of light and dark. If we think back to treating images as functions,  $F(x, y)$ , we can think of the gradient as a derivative operation  $F'(x, y)$ . Where the derivative is a measurement of intensity change.

## Sobel filters

The Sobel filter is very commonly used in edge detection and in finding patterns in intensity in an image. Applying a Sobel filter to an image is a way of **taking (an approximation) of the derivative of the image** in the  $xx$  or  $yy$  direction. The operators for  $Sobel_x$  and  $Sobel_y$ , respectively, look like this:

There are many different edge detection methods, the majority of which can be grouped into two categories:

- 1- Gradient
- 2- Laplacian.

The gradient method detects the edges by looking for the maximum and minimum in the first derivative of the image.

The Laplacian (Sobel filter) method searches for zero crossings in the second derivative of the image.

## Magnitude

Sobel also detects which edges are *strongest*. This is encapsulated by the **magnitude** of the gradient; the greater the magnitude, the stronger the edge is. The magnitude, or absolute value, of the gradient is just the square root of the squares of the individual  $x$  and  $y$  gradients. For a gradient in both the  $x$  and  $y$  directions, the magnitude is the square root of the sum of the squares.

$$\text{abs\_sobel}_x = \sqrt{(\text{sobel}_x)^2}$$

$$\text{abs\_sobel}_y = \sqrt{(\text{sobel}_y)^2}$$

$$\text{abs\_sobel}_{xy} = \sqrt{(\text{sobel}_x)^2 + (\text{sobel}_y)^2}$$

## Direction

In many cases, it will be useful to look for edges in a particular orientation. For example, we may want to find lines that only angle upwards or point left. By calculating the direction of the image gradient in the  $x$  and  $y$  directions separately, we can determine the direction of that gradient!

The direction of the gradient is simply the inverse tangent (arctangent) of the  $y$  gradient divided by the  $x$  gradient:

$$\tan^{-1}(\text{sobel}_y / \text{sobel}_x).$$



# Smoothing Images

## Learn to:

- Blur images with various low pass filters
- Apply custom-made filters to images (2D convolution)

## 2D Convolution ( Image Filtering )

As for one-dimensional signals, images also can be filtered with various **low-pass filters (LPF)**, **high-pass filters (HPF)**, etc. A **LPF helps in removing noise**, or **blurring** the image. A **HPF filters** helps in finding edges in an image.

OpenCV provides a function, **cv2.filter2D()**, to convolve a kernel with an image. As an example, we will try an averaging filter on an image. A 5x5 averaging filter kernel can be defined as follows:

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('opencv_logo.png')

kernel = np.ones((5,5),np.float32)/25
dst = cv2.filter2D(img,-1,kernel)

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([], plt.yticks([]))
plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
plt.xticks([], plt.yticks([]))
plt.show()
```

# image Blurring (Image Smoothing)

Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise. It actually removes high frequency content (e.g: noise, edges) from the image resulting in edges being blurred when this filter is applied. (Well, there are blurring techniques which do not blur edges). OpenCV provides mainly four types of blurring techniques.

## 1. Averaging

This is done by convolving the image with a normalized box filter. It simply takes the average of all the pixels under kernel area and replaces the central element with this average. This is done by the function **cv2.blur()** or **cv2.boxFilter()**. Check the docs for more details about the kernel. We should specify the width and height of kernel. A 3x3 normalized box filter would look like this:

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

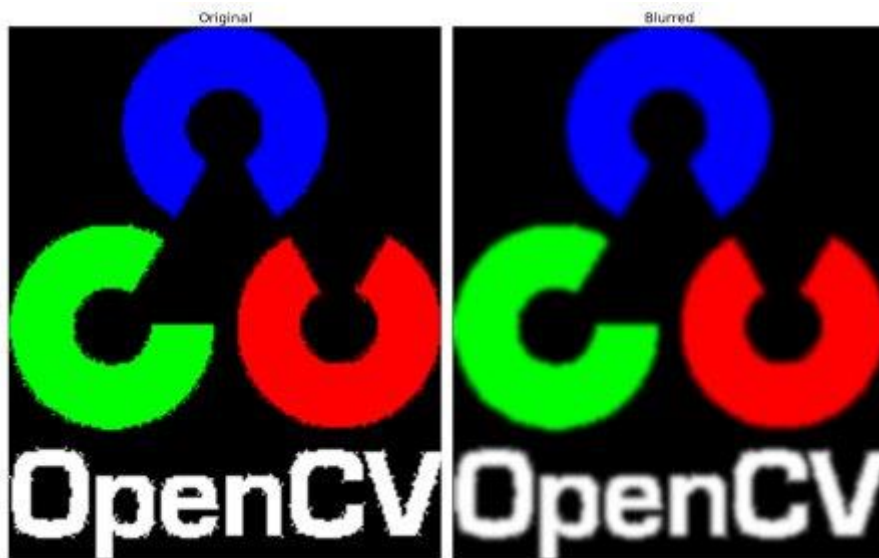
## 2. Gaussian Filtering

In this approach, instead of a box filter consisting of equal filter coefficients, a Gaussian kernel is used. It is done with the function, **cv2.GaussianBlur()**. We should specify the width and height of the kernel which should be **positive and odd**. We also should specify the standard deviation in the X and Y **directions**, sigmaX and sigmaY respectively. If **only sigmaX** is specified, sigmaY is taken as equal to sigmaX. If both are given as zeros, **they are calculated from the kernel size**. Gaussian filtering is highly effective in **removing Gaussian noise** from the image.

If you want, you can create a Gaussian kernel with the function, **cv2.getGaussianKernel()**.

The above code can be modified for Gaussian blurring:

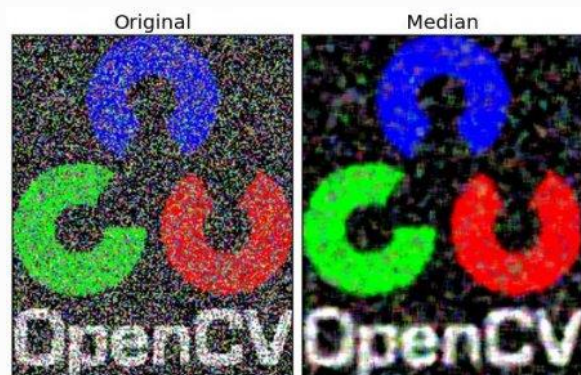
```
blur = cv2.GaussianBlur(img,(5,5),0)
```



### 3. Median Filtering

Here, the function **cv2.medianBlur()** computes the median of all the pixels under the kernel window and the central pixel is replaced with this median value. This is highly effective in removing salt-and-pepper noise. One interesting thing to note is that, in the Gaussian and box filters, the filtered value for the central element can be a value which may not exist in the original image. However this is not the case in median filtering, since the central element is always replaced by some pixel value in the image. This reduces the noise effectively. The kernel size must be a positive odd integer.

In this demo, we add a 50% noise to our original image and use a median filter. Check the result:



```
median = cv2.medianBlur(img,5)
```

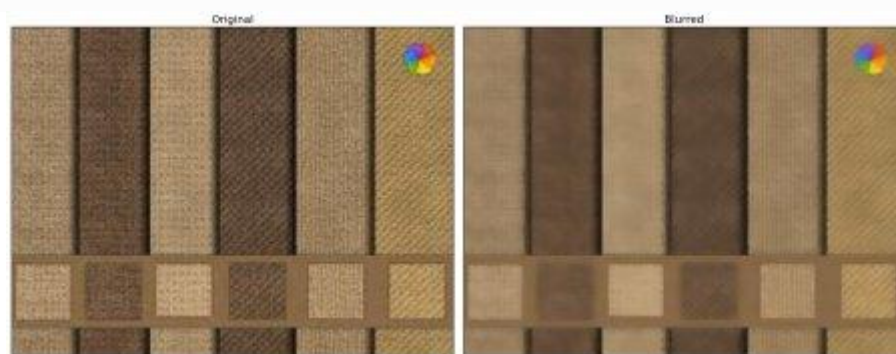
## Bilateral Filtering

As we noted, the filters we presented earlier tend to blur edges. This is not the case for the bilateral filter, `cv2.bilateralFilter()`, which was defined for, and is highly effective at noise removal while preserving edges. But the operation is slower compared to other filters. We already saw that a Gaussian filter takes the a neighborhood around the pixel and finds its Gaussian weighted average. This Gaussian filter is a function of space alone, that is, nearby pixels are considered while filtering. It does not consider whether pixels have almost the same intensity value and does not consider whether the pixel lies on an edge or not. The resulting effect is that Gaussian filters tend to blur edges, which is undesirable.

The bilateral filter also uses a Gaussian filter in the space domain, but it also uses one more (multiplicative) Gaussian filter component which is a function of pixel intensity differences. The Gaussian function of space makes sure that only pixels are 'spatial neighbors' are considered for filtering, while the Gaussian component applied in the intensity domain (a Gaussian function of intensity differences) ensures that only those pixels with intensities similar to that of the central pixel ('intensity neighbors') are included to compute the blurred intensity value. As a result, this method preserves edges, since for pixels lying near edges, neighboring pixels placed on the other side of the edge, and therefore exhibiting large intensity variations when compared to the central pixel, will not be included for blurring.

The sample below demonstrates the use of bilateral filtering (For details on arguments, see the OpenCV docs).

```
blur = cv2.bilateralFilter(img,9,75,75)
```



## Low pass Filter

Noise is generally seen as speckles or discolorations in an image and it does not contain useful information

- 1- Blur smooth an image
- 2- Block high-frequency parts of an image

It is reduce the high frequency noise

## LOW-PASS FILTERS

1	1	1
1	1	1
1	1	1

Averaging Filter

Low pass filter => typically take an average and not a difference as high pass filters do

# LOW-PASS FILTERS

## Normalize

Divide the total value of the kernel so that the components add up to **1**

$$\frac{1}{9} \times$$

1	1	1
1	1	1
1	1	1

Averaging Filter

$$\frac{1}{9} \times (1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1) = 1$$

So we need to normalize by dividing the total value of the kernel by nine.

$$\frac{1}{9} \times$$

1	1	1
1	1	1
1	1	1

$$\frac{1}{9} \times$$

80	80	100
100	40	105
50	90	120



this case, we're multiplying all the values by one and summing them up.

## Why Gaussian is the best ?

- 1- Blur / smooth and image
- 2- Block high frequency parts of an image
- 3- Preserve edges

1/ 16 x in kernel

1	2	1
2	4	2
1	2	1

## Gaussian Blur

```
image_brain = cv2.imread('H:\\Udacity - Computer Vision Nanodegree v1.0.0\\GitHub\\CVND_Exercises\\1_2_Convolutional_Filters_Edge_Detection\\images\\brain_MR.jpg')
image_copy_brain = np.copy(image_brain)
image_copy_brain = cv2.cvtColor(image_copy_brain , cv2.COLOR_BGR2RGB)
plt.imshow(image_copy_brain)

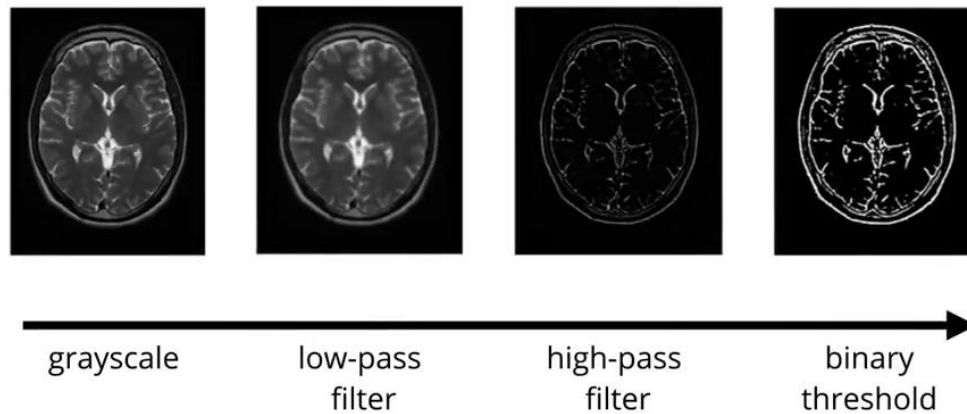
# Applying a Gaussian blur filter
# Before you must convert to grayscale
gray = cv2.cvtColor(image_copy_brain , cv2.COLOR_RGB2GRAY)
# Get the picture in gray mode
# the size of kernel is 5 x 5
# Standard Deviation -> if i set zero it will be automatically zero
gray_blur = cv2.GaussianBlur(gray ,(5,5) , 0 )

f , (ax1 , ax2 ) = plt.subplots(1 , 2 ,figsize=(20,10))

ax1.set_title('original gray ')
ax1.imshow(gray , cmap='gray')

ax2.set_title('blured image ')
ax2.imshow( gray_blur , cmap='gray')
```

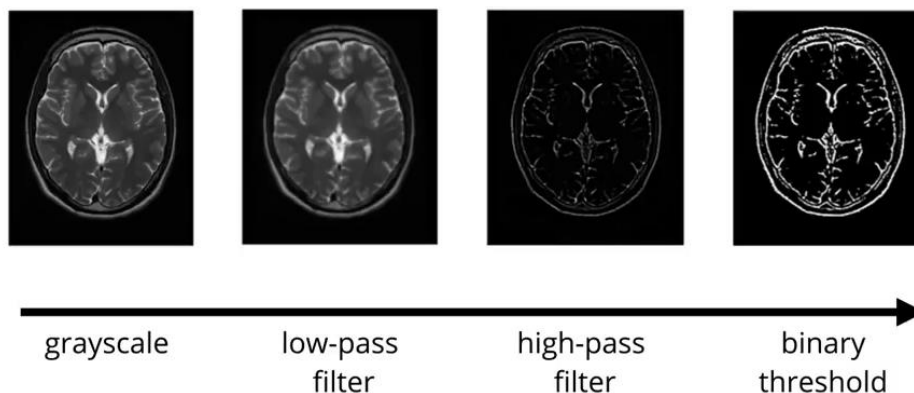
## CANY Edge detection



both low pass and high pass filters for accurate edge detection.

## EDGE DETECTION

What level of intensity change constitutes an edge?  
How can we consistently represent thick and thin edges?



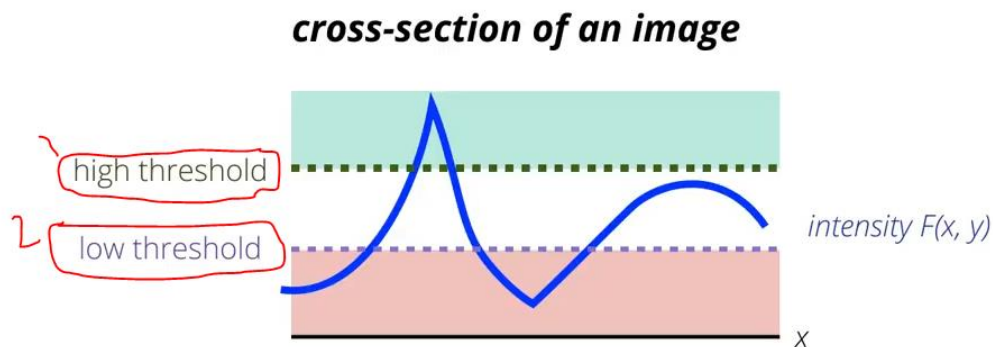
Canny edge detection is used widely in computer vision applications =>

Through a series of steps that consistently produces accuracy detect edges



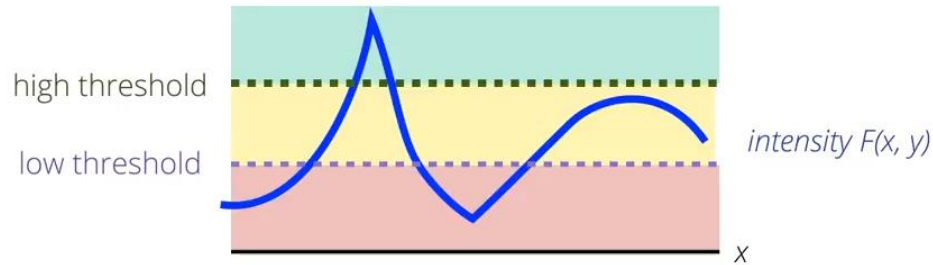
## Steps

- 1- Filters out of noise using a Gaussian blur
- 2- Find the strength and direction of edges using sobel filter
- 3- Using output of sobel filter canny then applies non-maximum suppression which look at the strength and direction of each detection edge and selects local maximum pixel create consistent one pixel wide thin that align with the strongest edges (to isolate the strongest edge and thin them one-pixel wide lines )
- 4- Finally using it uses a process called hysteresis thresholding to isolate the best edges (it is double thresholding process )



Any edge below this low threshold is considered weak and discarded

### ***cross-section of an image***



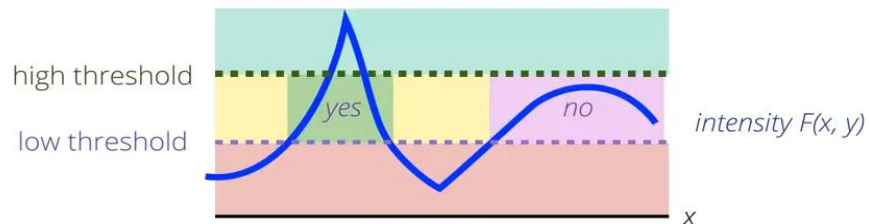
the high threshold will be kept only when  
it's connected to another strong edge.

- It will be kept only when it's connected to another strong edge

## CANNY EDGE DETECTION

- Eliminates weak edges and noise
- Isolates edges that are part of an object boundary

### ***cross-section of an image***



This way Canny eliminates weak edges and noise and isolates the edges

- 1- Eliminates weak edges and noise
- 2- Isolates edges that are part of an object boundary
- 3- This way Canny eliminate weak edges and noise and isolated the edge that are the most connected and therefore are most likely part of an object

boundary and because canny emphasizes important edges we will see that it's very useful in boundary and [ shape detection ]

### Important resources and display image

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

import cv2
import numpy as np

# task for canny edge detector
%matplotlib inline

path = 'H:\\Udacity - Computer Vision Nanodegree
v1.0.0\\GitHub\\CVND_Exercises\\1_2_Convolutional_Filters_Edge_Detection\\i
mages\\sunflower.jpg'

# Read in the image sunflower.jpg
image = mpimg.imread(path)

#plt.imshow(image)

# Copy of image
image_copy = np.copy(image)

# Convert to grayscale for filtering
gray_image = cv2.cvtColor(image_copy, cv2.COLOR_BGR2RGB)

# Show the output of image
plt.imshow(gray_image)
```

## 17. Shape Detection

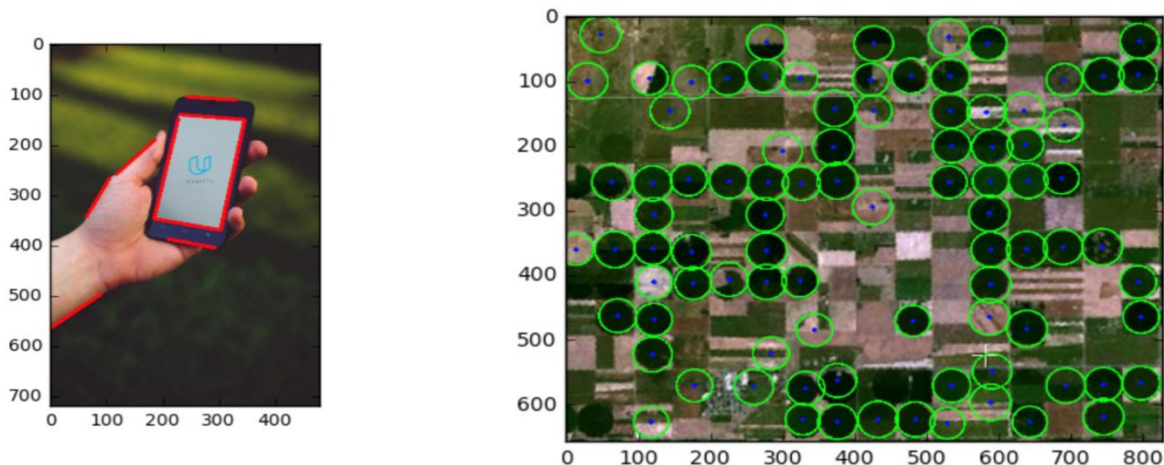
### Edge Detection

Now that you've seen how to define and use image filters for smoothing images and detecting the edges (high-frequency) components of objects in an image, let's move one step further. The next few videos will be all about how we can use what [we know about pattern recognition in images to begin identifying unique shapes and then objects](#).

#### Edges to Boundaries and Shapes

We know how to detect the edges of objects in images, but how can we begin to find [unifying boundaries around objects](#)? We'll want to be able to do this to separate and [locate multiple objects in a given image](#). Next, we'll discuss the Hough transform, which transforms image data from [the x-y coordinate system into Hough space](#), where you can easily [identify simple boundaries like lines and circles](#).

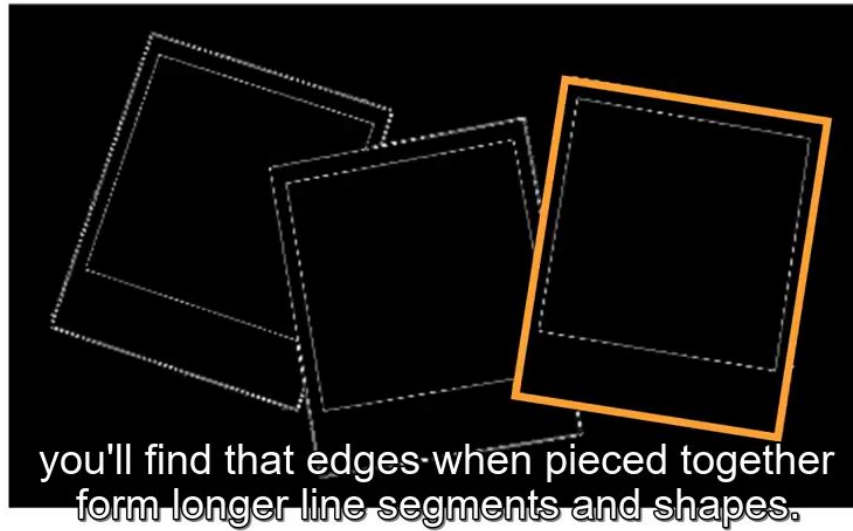
The Hough transform is used in a [variety of shape-recognition applications](#), as seen in the images pictured below. On the left you see how a Hough transform can find the edges of a phone screen and on the right you see how it's applied to an aerial image of farms (green circles in this image).



[Hough transform](#) applied to phone-edge and circular farm recognition.

# LINE DETECTION

Complex boundaries are often made of several lines

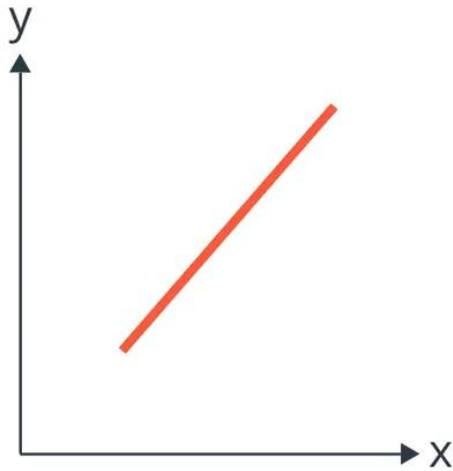


When u need to edge detection

- U can represent any line as a function of space
- In image space the equation represent a line  $y = mx + b$

# LINE DETECTION

Can represent any line as a function of space



$$y = m_0x + b_0$$

$m$  = slope of the line

$b$  = how much it's shifted in the y-direction

$M \Rightarrow$  is slope

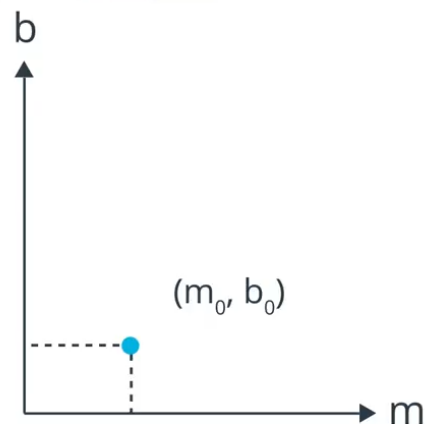
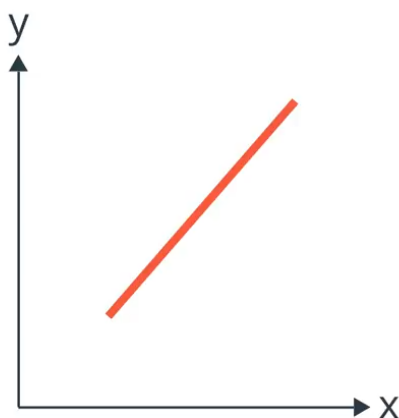
$x_0 = b_0 \Rightarrow$  it is constant

$b$  = how much it's shifted in the y direction

A useful transformation is moving this line representation from image space to Hough space also called

Parameter space

The Hough transform converts a line in image space to a point in Hough space

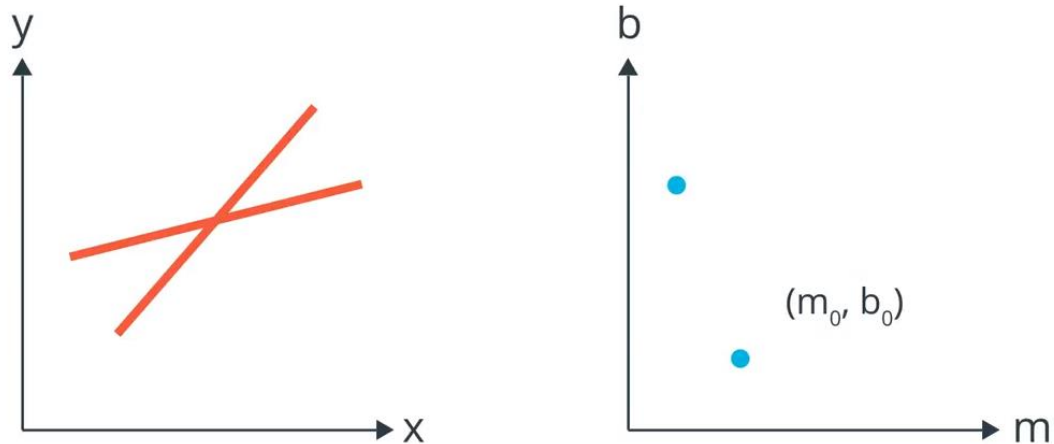


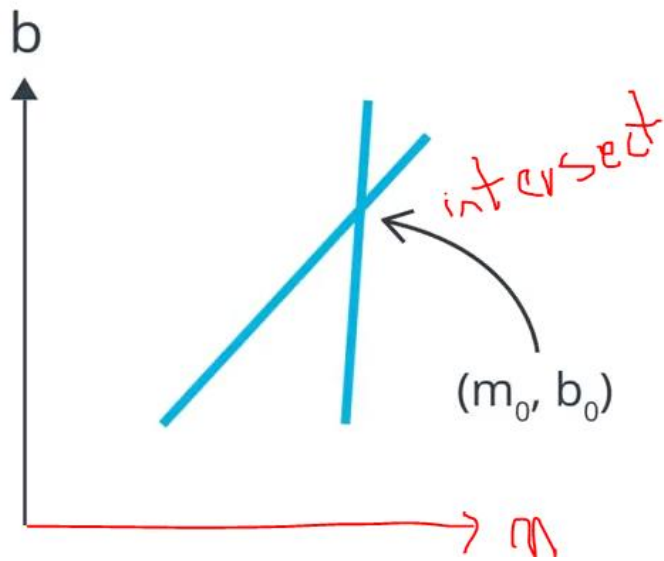
The Hough transform converts data points from image space to Hough space and represents a line in the simplest way as a point at the coordinate  $m_0$  and  $b_0$

- Multiple line represented by multiple  $m$  and  $b$  coordinates in Hough space
- And pattern in Hough space can help us identify lines or other shapes

## HOUGH SPACE

The Hough transform converts a line in image space to a point in Hough space

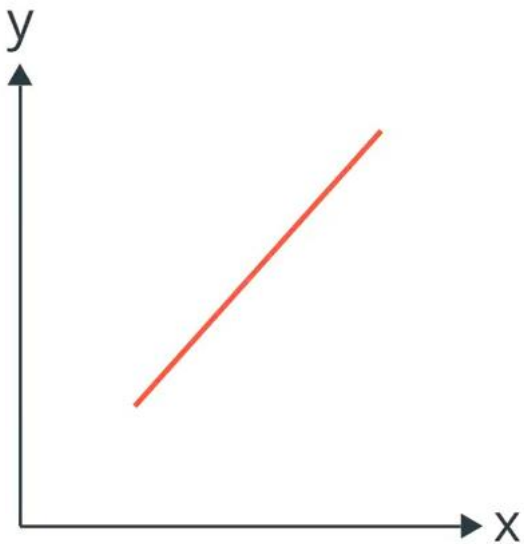




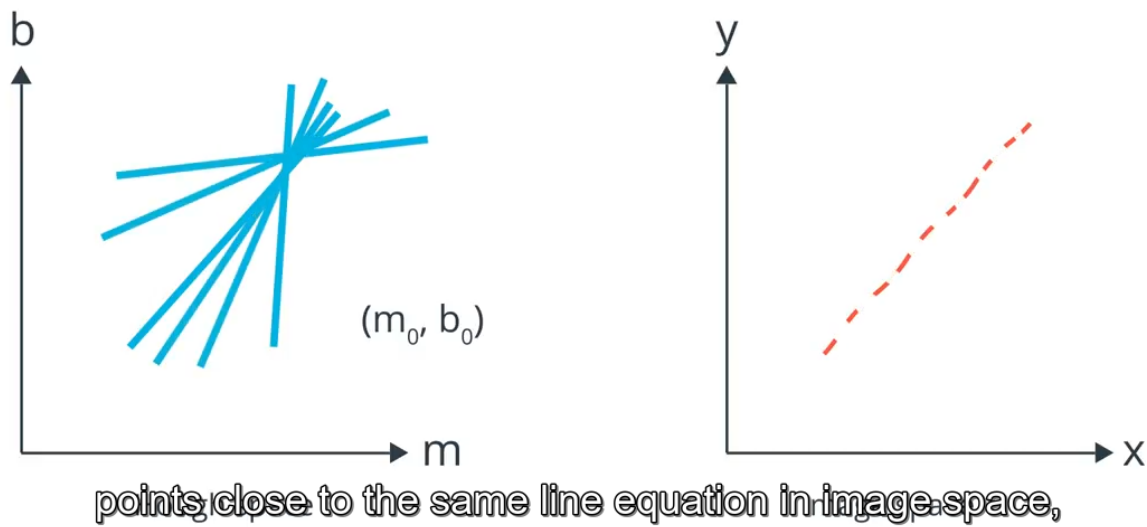
For example :

Look at the two lines in Hough space that intersect at the point  $m_0$  and  $b_0$  ?

It is the line of equation  $y = m_0 x + b_0$







And if u have a line made of line made mini segments or points close to the same line equation in image space

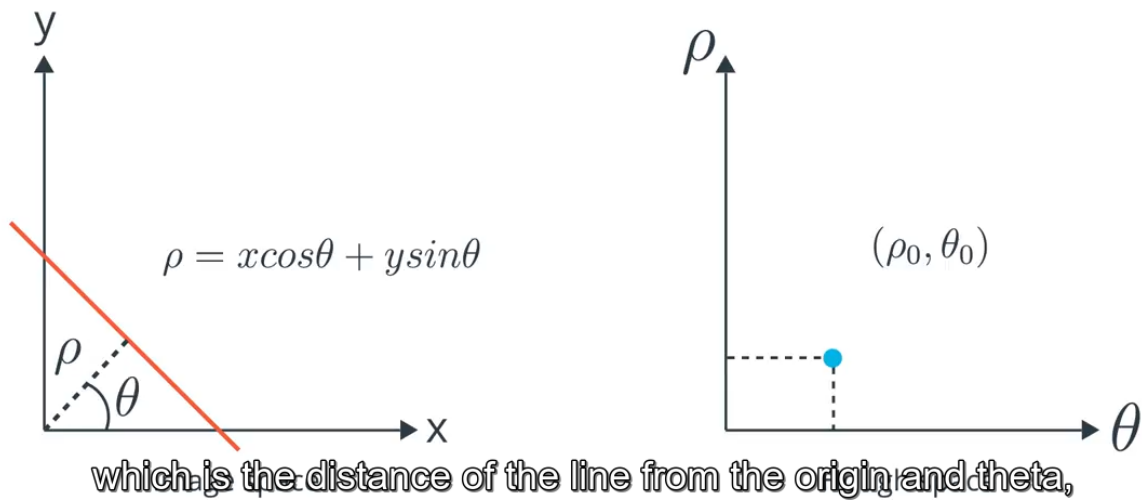
This turn into many interesting lines in Hough space

Imagine this line as part of an edge detected image where a line just has some small discontinuities in it

Our strategy then for finding continues lines in image is to look at intersection points in Hough space this is simplified example you may have thought about the case of a line straight up and down which technically has an infinite slope so better way to transform the image space is by turning Hough space into polar coordinates

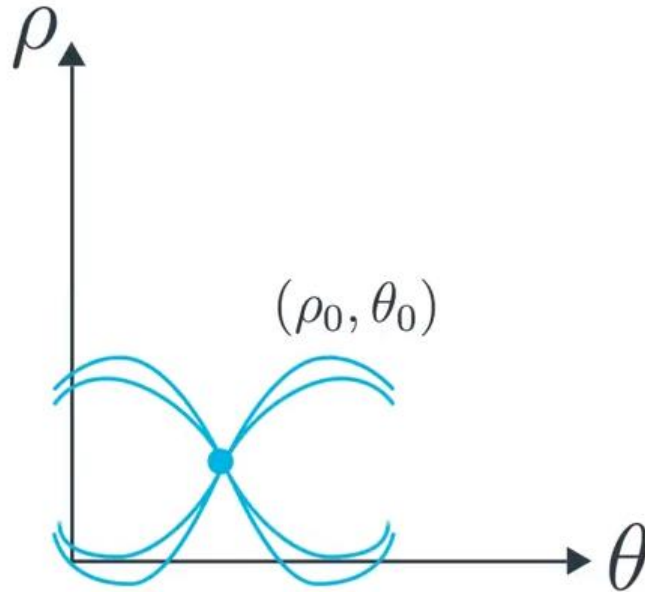
# HOUGH SPACE

$\rho$  = the distance of the line from the origin



Instead of m and b we have rho p = the distance of line from the origin and theta

Theta => the angle from the horizontal axis



Now fragmented line or points that fall in a line can be identified in Hough space as the intersections of sinusoids

Know , Let`s your knowledge of Hough space then walk through how to use this transform to identify lines

Problem in Hough space ?

Imagine the point in vertical , it indicate to draw a finite points

Using Polar representation for lines ...

vertical بيطلع قيم سيناسويد عمرها ماhtعمل قيم

randomization طب الحجات اللي فيها

**Extension 1** to use image gradient ...

**Extension 2** give more votes for stronger edges (use magnitude of gradient )

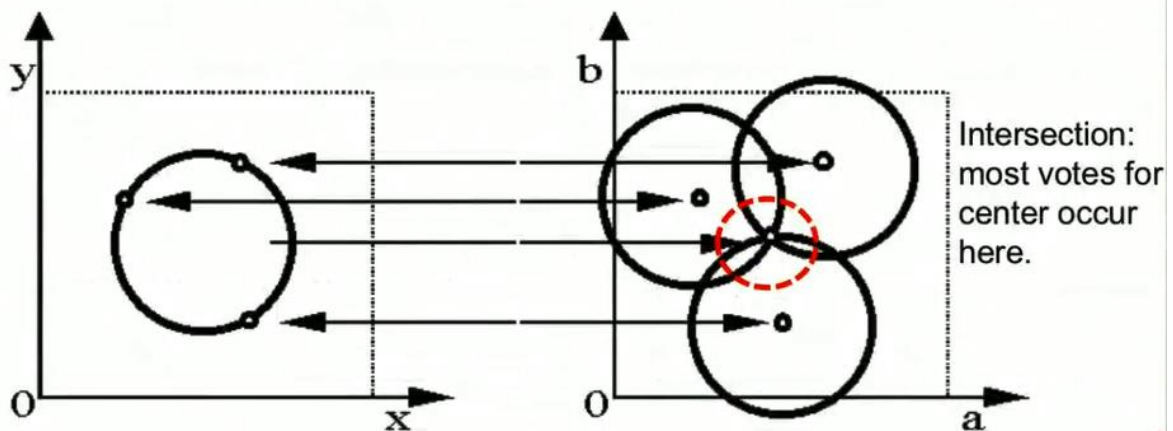
**Extension 3** change the sampling of (d , 0 ) to give more less resolution

**Extension 4** the same procedure can be used with circles squares or any other shapes

- 
- Circle: center (a,b) and radius r

$$(x_i - a)^2 + (y_i - b)^2 = r^2$$

- For a fixed radius r, unknown gradient direction





- After Coding and get line return in array
- You can see it creates a number of lines around the screen and on the hand
- They actually might want to filter out some of these shorter lines

```
# increase min_line_length = 100 => it is the minimum number of Hough space to intersections it takes to find a line
```

```
rho = 1
```

```
theta = np.pi/180
```

```
threshold = 60
```

```
min_line_length = 100
```

```
max_line_gap = 5
```

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

```
# Next Step make a copy an image
```

```
#
```

```
line_image = np.copy(image_copy)
```

```
for line in lines:
```

```
    for x1,y1,x2,y2 in line:
```

```
        cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0) , 5 )
```

```
plt.imshow(line_image )
```



Finding continuous lines like this can be useful in helping from boundaries and in noticing details about object structure

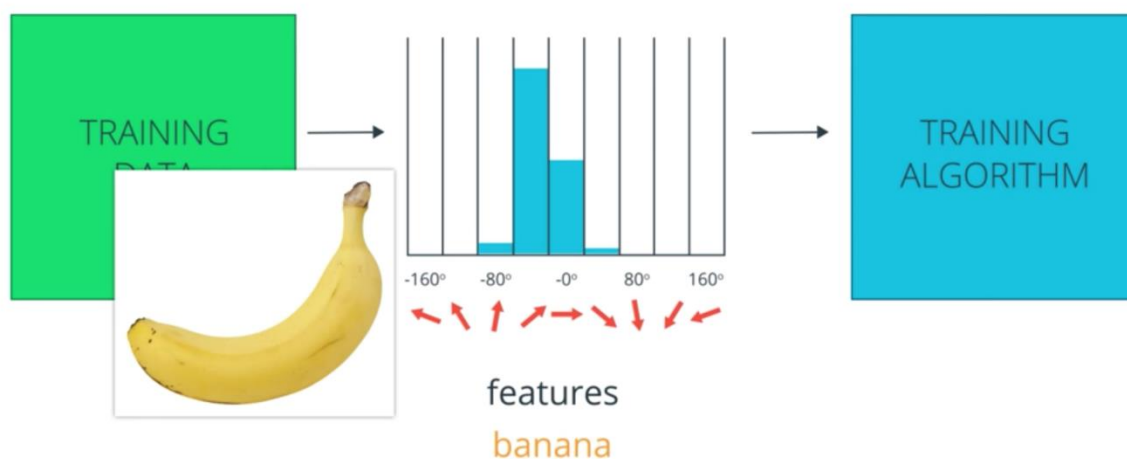
In many cases , you will have to decide between using image contours or Hough space to find object boundaries

Next : Hough space and learn about other segmentation techniques

## Feature Extraction and Object Recognition

So, you've seen how to detect consistent shapes with something like the Hough transform that transforms shapes in x-y coordinate space into intersecting lines in Hough space. You've also gotten experience programming your own image filters to perform edge detection. Filtering images is a feature extraction technique because it filters out unwanted image information and extracts unique and identifying features like edges or corners.

Extracting features and patterns in image data, using things like image filters, is the basis for many object recognition techniques. In the image below, we see a classification pipeline that is looking at an image of a banana; the image first goes through some filters and processing steps to form a feature that represent that banana, and this is used to help classify it. And we'll learn more about feature types and extraction methods in the next couple lessons.

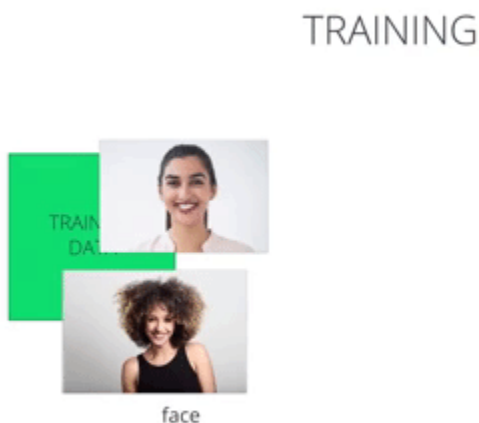


Training data (an image of a banana) going through some feature extraction and classification steps.

## Haar Cascade and Face Recognition

In the next video, we'll see how we can use a feature-based classifier to do face recognition.

The method we'll be looking at is called a **Haar cascade classifier**. It's a machine learning based approach where a cascade function is trained to solve a **binary classification** problem: face or not-face; it trains on a lot of positive (face) and negative (not-face) images, as seen below.



After the classifier sees an image **of a face or not-face**, it extracts features from it. For this, Haar filters shown in the below image are used. **They are just like the image filters you've programmed!** A new, filtered image is produced when the input image is convolved with one of these filters at a time.

**Next, let's learn more about this algorithm and implement a face detector in code!**

From each image, it detects so called haar features

Haar feature : are gradient measurements that look at rectangle regions around a certain pixel area and somewhat subtract these areas to calculate a pixel difference

- 1- Certain pixel area
- 2- Subtract these area to calculate the a pixel difference

This is similar to how convolution kernels work only at larger scale

Haar features detect patterns like

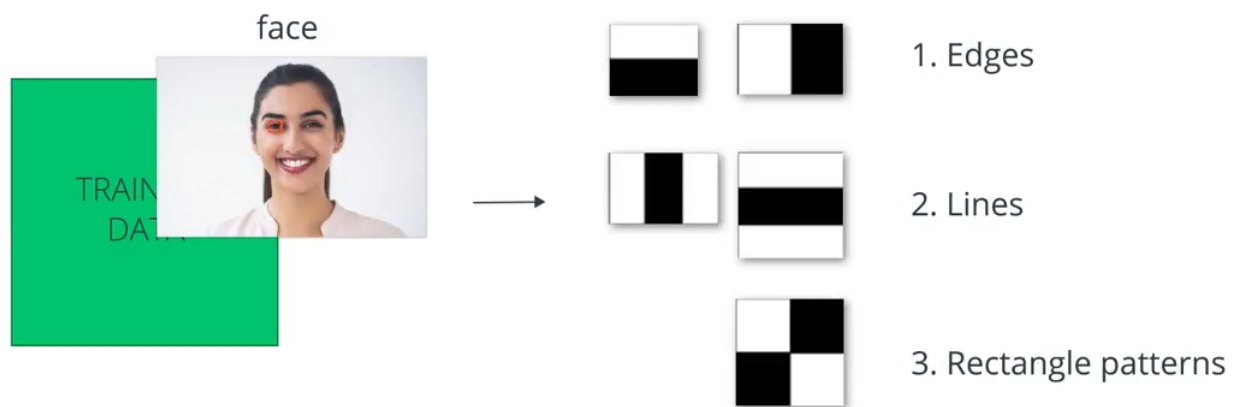
- 1- Edges

- 2- Lines
- 3- And more complex rectangular patterns

In the case of face detection , line and rectangle are especially useful features because pattern of alternating bright and dark areas define a lot of feature on a face

For example , our pupils are typically a very dark feature and cheeks and chins define high gradient outline for a face

## HAAR FEATURES



Haar features effectively identify extreme areas of bright and dark on a face

So haar feature are looking pretty similar to the beginning steps of a convolutional neural network

Or even a HOG feature extractions

It's a series of cascade

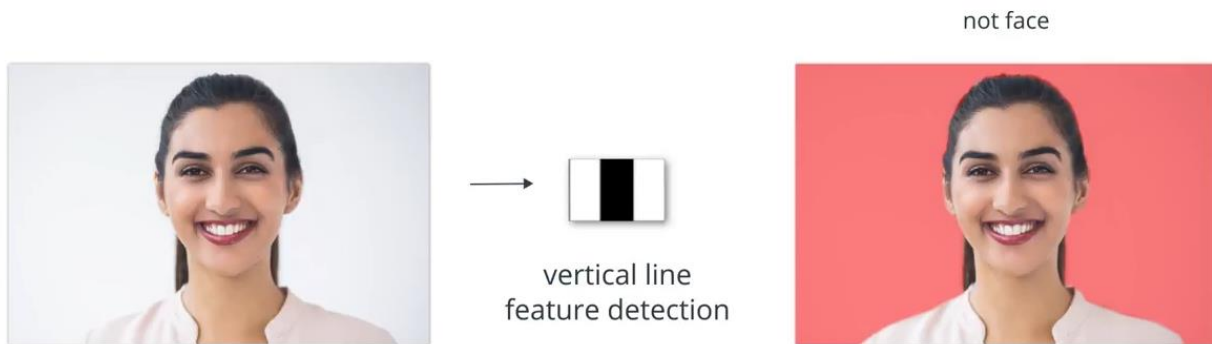
For any image of a face : a large portion of this image will be a non face region so what haar cascade do

It look like a image and applies a haar feature detector like a vertical line detector and perform an then perform classifications on the entire image

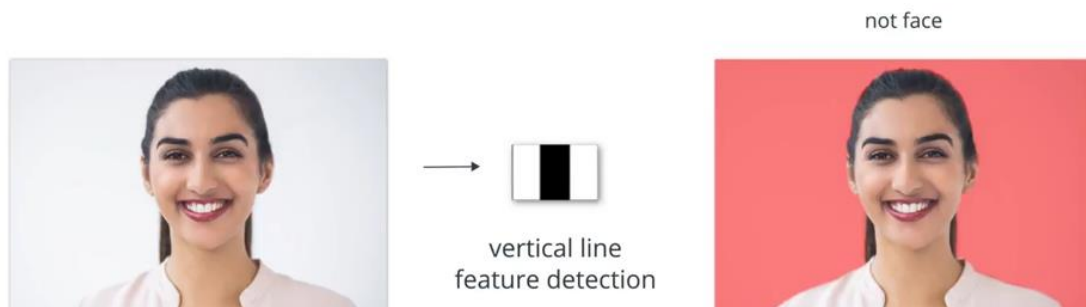


- If it doesn't get enough of a feature detections response => a classifier an area of an image as not aface and discard the information

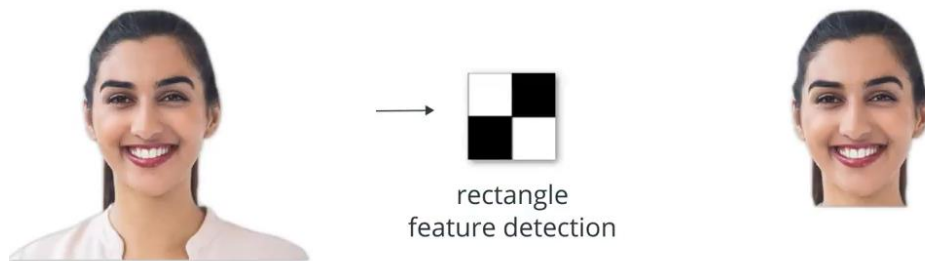
## HAAR CASCADES



- Then it feeds this reduced image area to (rectangle feature detection ) the next feature detector and classifier the image again discarding irrelevant non-face areas at every steps this is called 'cascade of classifier'



- Then it feeds this reduced image area to (rectangle feature detection ) the next feature detector and classifier the image again |

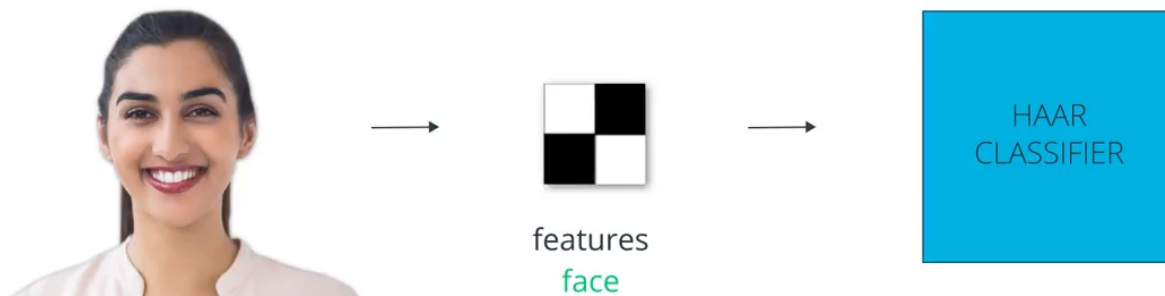


and recognizing only the area in an image that's been

Quickly getting rid of irrelevant information image data makes this algorithm very fast, fast enough for processing a video stream in real time on a laptop on a computer

## HAAR CASCADES

Cascade of classifiers



Haar cascade may also be used in selecting an area of interest for further processing so let's get more practice with haar cascading

## Face Recognition and the Dangers of Bias





### Algorithms with Human and Data Bias

Most of the models you've seen and/or programmed, rely on large sets of data to train and learn. When you approach a challenge, it's up to you as a programmer, to define

functions and a model for classifying image data. Programmers and data define how classification algorithms like face recognition work.

It's important to note that both data and humans come with their own biases, with unevenly distributed image types or personal preferences, respectively. And it's important to note that these biases propagate into the creation of algorithms. If we consider face recognition, think about the case in which a model like a Haar Cascade is trained on faces that are mainly white and female; this network will then excel at detecting those kinds of faces but not others. If this model is meant for general face recognition, then the biased data has ended up creating a biased model, and algorithms that do not reflect the diversity of the users it aims to serve is not very useful at all.

The computer scientist, [Joy Buolamwini](#), based out of the MIT Media Lab, has studied bias in decision-making algorithms, and her work has revealed some of the extent of this problem. One study looked at the error rates of facial recognition programs for women by shades of skin color; results pictured below.

						
	TYPE I	TYPE II	TYPE III	TYPE IV	TYPE V	TYPE VI
	1.7%	1.1%	3.3%	0%	23.2%	25.0%
	5.1%	7.4%	8.2%	8.3%	33.3%	<b>46.8%</b>
	11.9%	9.7%	8.2%	13.9%	32.4%	<b>46.5%</b>

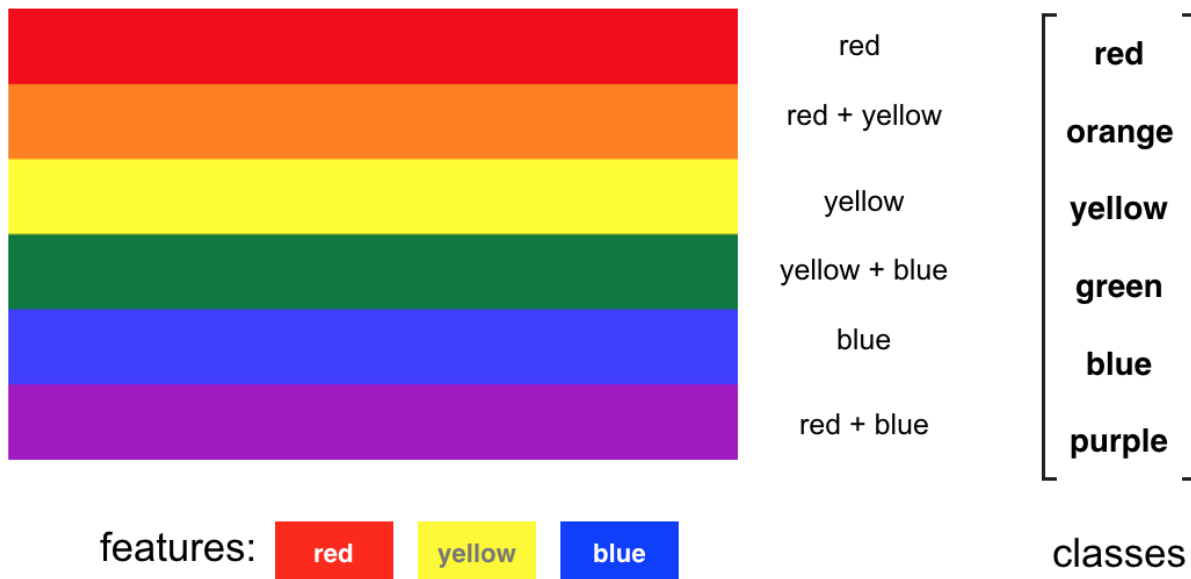
## 26. Beyond Edges, Selecting Different Features

### Features

**Features** and **feature extraction** is the basis for many computer vision applications. The idea is that any **set of data, such as a set of images**, can be represented by a **smaller, simpler model** made of a combination of visual features: a few colors and shapes. (This is true with one exception: completely random data!)

If you can find a good model for any set of data, then you can start to find ways to **identify patterns** in data based on **similarities** and **differences** in the features in an image. This is especially important when we get to deep learning models for **image classification**, which you'll see soon.

Below is an example of a simple model for rainbow colors. Each of the colors below is actually a combination of a smaller set of color features: red, yellow, and blue. For example, purple = red + blue. And these simple features give us a way to represent a variety of colors and classify them according to their red, yellow, and blue components!



# Types of Features

We've described features as measurable pieces of data in an image that help distinguish between different classes of images.

There are two main types of features:

1. Color-based and
2. Shape-based

Both of these are useful in different cases and they are often powerful together. We know that color is all you need should you want **to classify day/night images** or **implement a green screen**. Let's look at another example: say I wanted to classify a **stop sign vs. any other traffic sign**. Stop signs are *supposed* to stand out in color and shape! A stop sign is an octagon (it has 8 flat sides) and it is very red. Its red color is often enough to distinguish it, but the sign can be obscured by trees or other artifacts and the shape ends up being important, too.

As a different example, say I want to detect a face and perform **facial recognition**. I'll first want to detect a face in a given image; this means at least recognizing the boundaries and some features on that face, which are all determined by **shape**. **Specifically**, I'll want to identify the **edges of the face and the eyes and mouth on that face**, so that I can identify the face and recognize it. Color is not very useful in this case, but **shape is critical**.

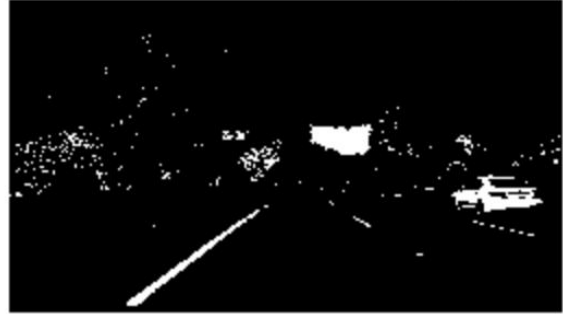
## A note on shape

**Edges are one of the simplest shapes** that you can detect; edges often define the boundaries between objects but they may not provide enough information to find and identify **small features on those objects (such as eyes on a face)** and in the next videos, we'll look at methods for **finding even more complex shapes**.

As you continue learning, keep in mind that selecting the right feature is an **important computer vision task**.

## Example Application: Lane Finding

You've already had some practice with this concept, but you can use feature/edge detection and color transforms to very effectively detect lane lines on a road. If you'd like to learn more about this technique, I suggest checking out [this blog post](#).



Identifying edges and lane markings on a road.