TYPES OF FEATURES

1. EDGES
2. CORNERS
3. BLOBS

Edges ➔ Area with a high intensity gradient

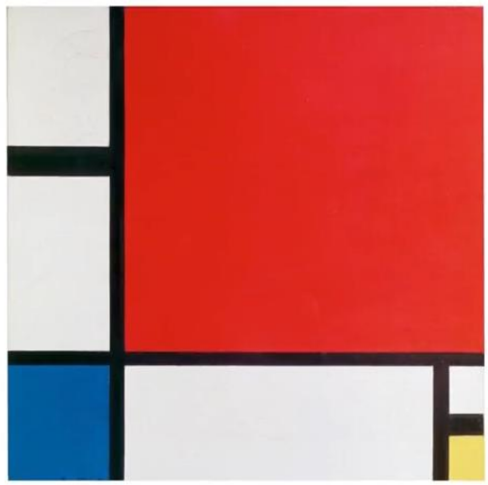
Corners ➔ At the intersection of two edges it is sharp point

Blobs ➔ region-based feature areas of extreme brightness or unique texture (extreme highs or low in intensity or areas of a unique texture)

We will most interested to detecting coroners because because it repeatable feature which means it easy to recognize given two or more images of the same scene

Example ➔

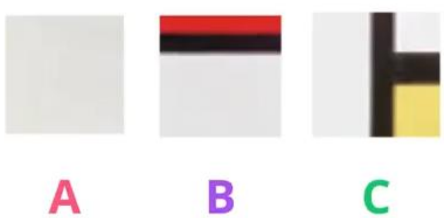
TYPES OF FEATURES



Mondrian painting

Three patches to look like what area do they match ?

PATCHES



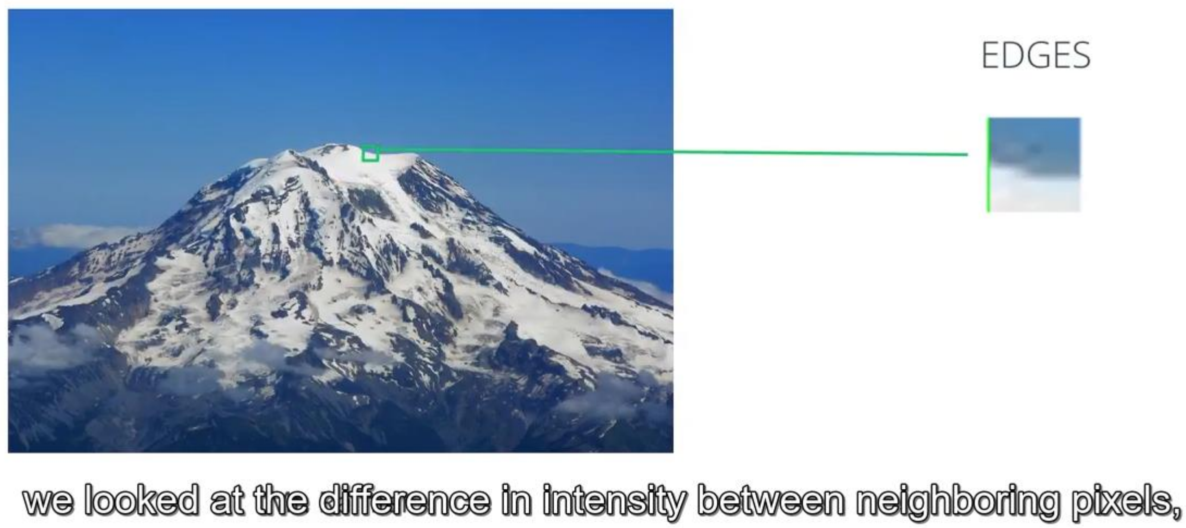
Can you tell me what a rectangular area they match with on this image ?

Patch A ➔ just a single patch of color ant it matches with a lot of areas like this all around the rectangle so it is bad feature because it is not unique

Patch B ➔ its orientations that it matches with an edge at the bottom of the red rectangle but we can still move in this edge to the left and right and it would still match we can only approximate where the edge appears on the image it is very hard to get the exact locations

Patch C ➔ it is a corners it actually contain two corners and its locations is easily identified as the bottom right (this is because a corner represents a point where two edges change and if u move either of these up and down the corner patch will not match exactly with that area so corners are easiest to match and make good feature because they are so unique)

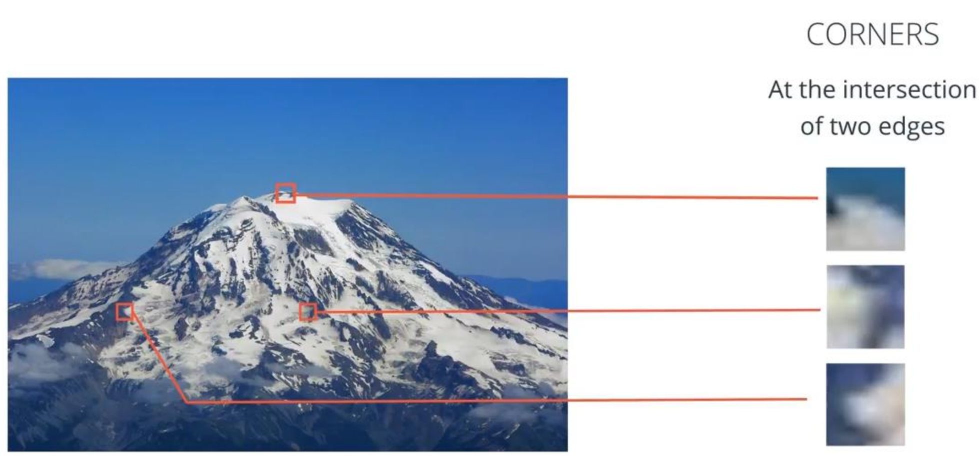
Corner Detection >



When a build an edge detector we looked at the difference in intensity between neighboring pixels and edge was detected if there was a big and abrupt change in intensity in any one direction up or down or left -right or diagonal

Recall the change in intensity in an image is also referred to as gradient and also detect corners by relying on these gradient measurements

We know the corners are of intersections two edges and we can detect them by taking a windows which is generally a square area that contains a groups of pixels and looking at the where the gradient is high all directions



CORNERS

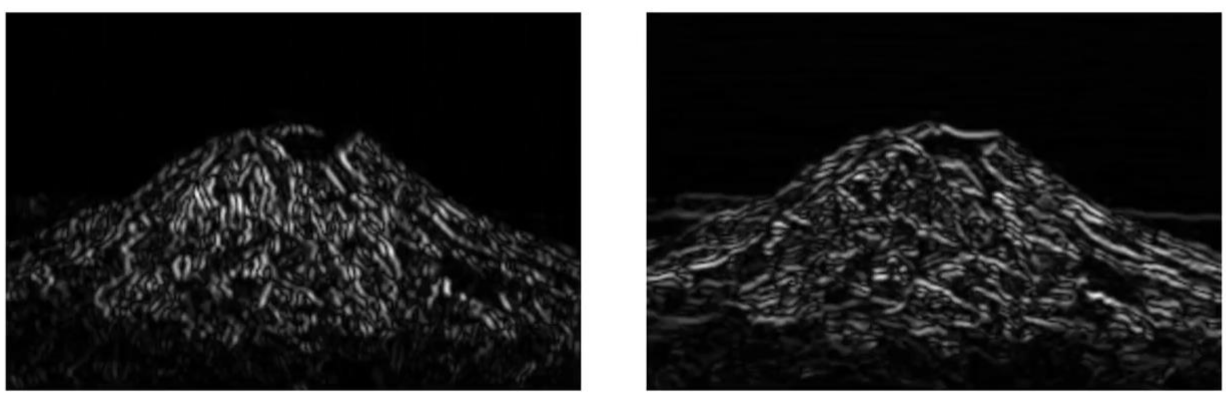
the intersection
of two edges

Each of these gradient measurement has an associated magnitude which is a measurement of the strength of the gradient and Directions which is the direction of the image of the change in intensity

And both of these values can be calculated as by sobel operators

Sobel operators take the intensity change or gradient of image in the x and y direction separately

GRADIENT



Sobel x Sobel y

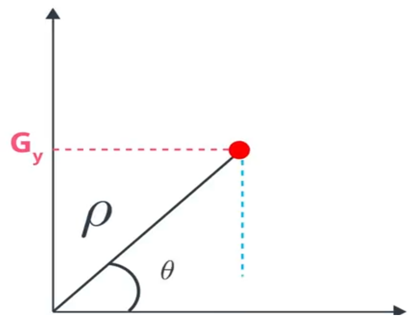
You may notice that these look a little different that our curl convolution before because they haven`t turned into a binary threshold of image

Then we want to get the magnitude and directions of the total gradient form these two values

To do that we actually convert these values from image space X Y to polar coordinates with magnitude rho and directions theta

GRADIENT

Convert G_x and G_y to polar coordinat



$$\rho = x cos\theta + y sin\theta$$

Any pixel location you can think of G_X and G_Y as the lengths of two sides of gradient triangle

G_X the length of the bottom side

G_Y the length of the right side

The total magnitude row of this gradient is the diagonal is the diagonal on this triangle or the square root of sum of these two gradient and the directions theta of the gradient is the calculated the inverse tangent of G_Y over G_X

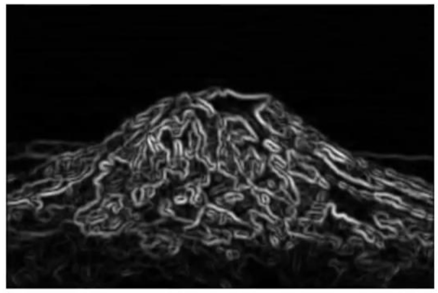
MAGNITUDE

$$\rho = \sqrt{G_x^2 G_y^2}$$

DIRECTION

$$\theta = tan^{-1}\frac{G_y}{G_x}$$

GRADIENT



Gradient Magnitude

The resulting gradient magnitude image should look something like this,

With the biggest gradients corresponding to the brightest lines

Now ? What mini corner detectors do is to take a window and shift it

1- Shift a window around an area in an image
2- Check for a big variation in the direction gradient and this large variation identifies a corner

I will be walking through coding a simpler corner detector that take advantage of this knowledge and finds corner`s based on identify locations with the largest variation in gradient for shifting window

```
# 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_3_Types_of_Features_Image_Segmentation\\images\\chessboard.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 ouput of image

plt.imshow(gray_image)

# Convert to grayscale for processing

gray = cv2.cvtColor(image_copy , cv2.COLOR_RGB2GRAY)


# Convert to float type

gray  = np.float32(gray)

# I will then convert these to floating point values that the harris coner detector will use

# Detect corners

# a Harris corner detector using to openCv function cornerHarris

# Function => cornerHarris()

# Take argument  grayscale float values

# The size of the neighborhood to look at when identify potential corners two means a two by two
pixel # square and since the corners are well marked in the example a small window like this will work
well 2


# Then take in the size of the sobel operator  three which typical size of kernel size  3


# And lastly a constant value that helps determine which points are considered corners 0.04 it is typical


# A slightly lower value for this constant will result in more corners detected and his produces an
output image i`ll call dst for destinations


dst = cv2.cornerHarris(gray , 2 , 3 , 0.04 , )
```

This image should have the corners marked as bright point and non-corners as darker pixel

darker pixel => non - corners

bright point => corners

#plt.imshow(gray , cmap='gray')

After Display

in this image , it`s actually very hard to see the bright corner points

So i`ll perform one more operationon these corners

Which will be to dilata them

Dialata corner image to enhance corner points

In Computer Vision dilation enlarges bright regions or regions in the # # foreground like thes corners so that we will be able to seen them better

dst = cv2.dilate(dst , None)

Display result

plt.imshow(gray , cmap='gray')

Now you can see the corners fairly well as these a bright points in the image

The last copule of steps will be to select and display the strongest cornsers

Select and Display strong corners

Define a threshold for extracting strong corners

This value may vary indepenting on the image

In this case i use the lower threshold at least one tenth ot the maximum corner detection value 0.1

threshold = 0.1 * dst.max()

Create an image copy to draw corners on

Corner_image = np.copy(image_copy)

Iterate through all the corners and draw them on the image (if they pass the threshold)

```
for j in range(0,dst.shape[0]):
    for i in range(0,dst.shape[1]):
        if(dst[j,i] > threshold ):
            # I'll draw the image
            # image , center pt , radius , color , thickness
            # Draw a small green circle on our strong corners on our image copy
            cv2.circle(Corner_image , (i , j) , 2  , (0,255,0) , 1 )



plt.imshow(corner_image)
```

We can see the most values of our corners were detected

We `re actually missing a couple right here and here

So u we may to change our threshold values change to 0.01

now u can see that all the corners on the chess board are detected

It`s actually pretty intersting to see where these green circle appear

the very bottem right corner of the board because ther`s no chnage in intensity '# But at every black and white intersection point we detect a corner '

you can imagine using these corner points to get information cbout the chessboard dimenstions or using a subset of these points to perform a perspective transformation

Corners alone can be useful for many types of analysis and geometric transformations

that helps determine which points are considered corners 0.04 it is typical

04. Dilation and Erosion

Dilation and erosion are known as **morphological operations**. They are often performed on binary images, similar to contour detection. Dilation enlarges bright, white areas in an image by adding pixels to the perceived boundaries of objects in that image. Erosion does the opposite: it removes pixels along object boundaries and shrinks the size of objects.

Often these two operations are performed in sequence to enhance important object traits!

Dilation

To dilate an image in OpenCV, you can use the `dilate` function and three inputs: an original binary image, a kernel that determines the size of the dilation (None will result in a default size), and a number of iterations to perform the dilation (typically = 1). In the below example, we have a 5x5 kernel of ones, which move over an image, like a filter, and turn a pixel white if any of its surrounding pixels are white in a 5x5 window! We'll use a simple image of the cursive letter "j" as an example.

1- original binary image
2- Kernel that determines the size of the dilation
3- number of iterations to perform the dilation

```
# Reads in a binary image
image = cv2.imread('j.png', 0)

# Create a 5x5 kernel of ones
kernel = np.ones((5,5),np.uint8)

# Dilate the image
dilation = cv2.dilate(image, kernel, iterations = 1)
```

Erosion

To erode an image, we do the same but with the `erode` function.

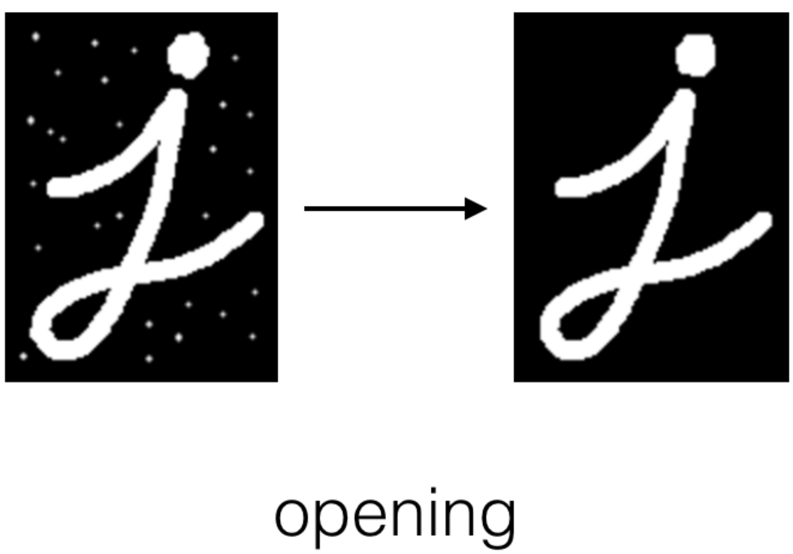
```
# Erode the image
erosion = cv2.erode(image, kernel, iterations = 1)
```

Opening

As mentioned, above, these operations are often *combined* for desired results! One such combination is called **opening**, which is **erosion followed by dilation**. This is useful in noise reduction in which erosion first gets rid of noise (and shrinks the object) then dilation enlarges the object again, but the noise will have disappeared from the previous erosion!

To implement this in OpenCV, we use the function `morphologyEx` with our original image, the operation we want to perform, and our kernel passed in

```
opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
```

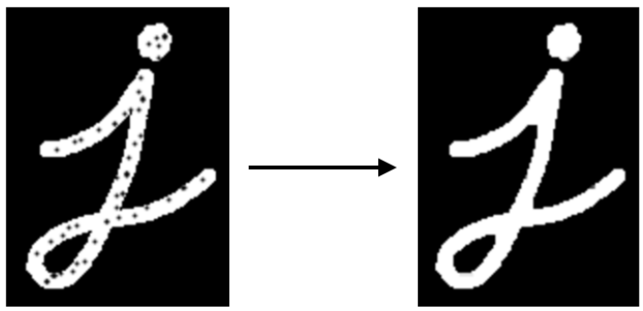


Closing

Closing is the reverse combination of opening; it's **dilation followed by erosion**, which is useful in *closing* small holes or dark areas within an object.

Closing is reverse of Opening, Dilation followed by Erosion. It is useful in closing small holes inside the foreground objects, or small black points on the object.

```
closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```



closing

Many of these operations try to extract better (less noisy) information about the shape of an object or enlarge important features, as in the case of corner detection!

Image Segmentation

Now that we are familiar with a few simple feature types, it may be useful to look at how we can group together different **parts of an image by using these features**. Grouping or segmenting images into distinct parts is known as image segmentation.

The simplest case for image segmentation is in background subtraction. In video and other applications, it is often the case that a human has to be isolated from a static or moving background, and so we have to use segmentation methods to distinguish these areas. Image segmentation is also used in a variety of complex recognition tasks, such as in classifying every pixel in an image of the road.

In the next few videos, we'll look at a couple ways to segment an image:

1. using contours to draw boundaries around different parts of an image, and
2. clustering image data by some measure of color or texture similarity.

IMAGE SEGEMENTATION

Partially-segmented image of a road; the image separates areas that contain a pedestrian from areas in the image that contain the street or cars.

https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_contours/py_table_of_contents_contours/py_table_of_contents_contours.html

Image Contours

Edge detection algorithm are used often used to detect the boundaries of objects

After , performing edge detection you will be often be left with sets of edges that highlight not only object boundaries but also interesting feature and lines and to image segmentation

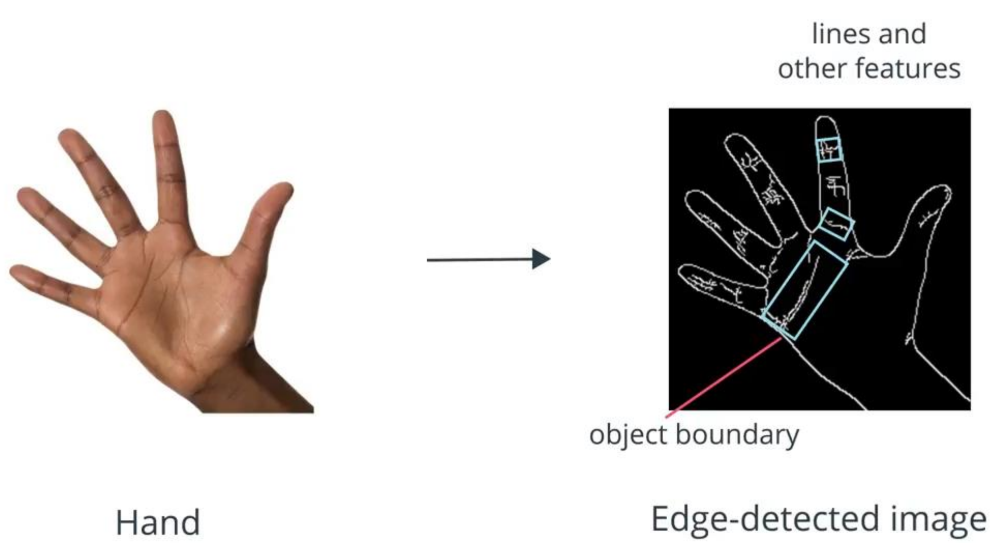


IMAGE SEGEMENTATION

You will want only complete closed boundaries that market distinct areas and object in an image

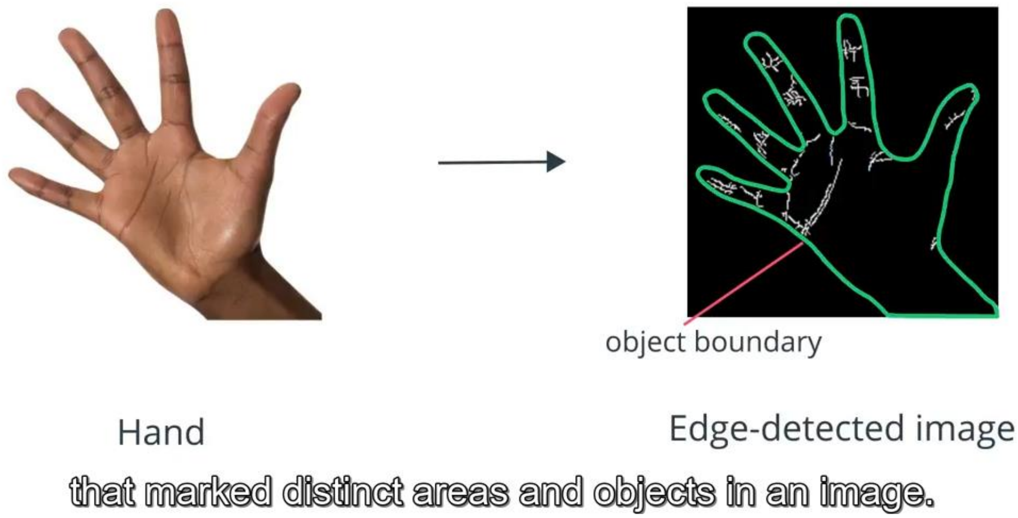
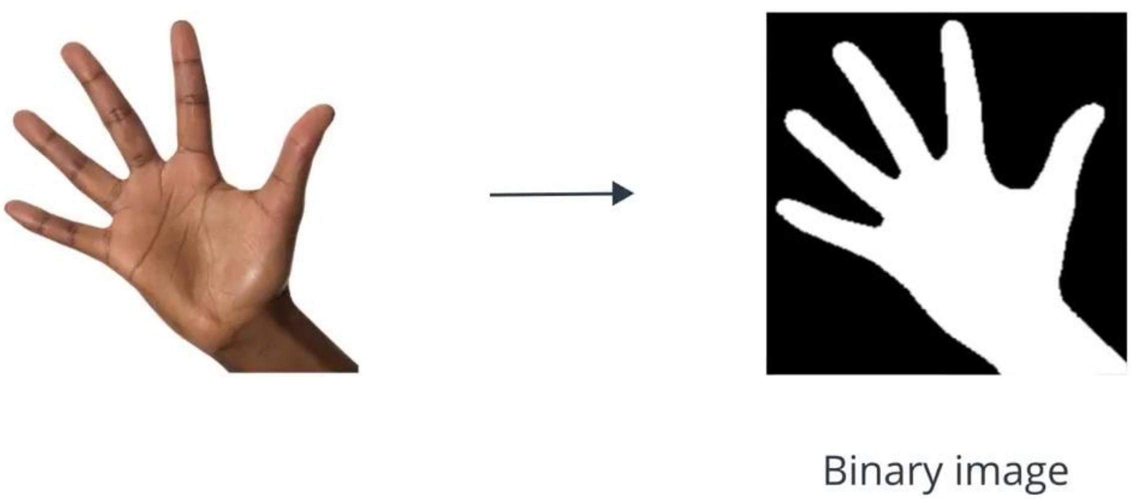


Image contour technique :D

Image contours are continuous curves that follow the edge along a perceived boundary

So can be used for image segmentation and they can also provide a lot information about the shape of an object boundary

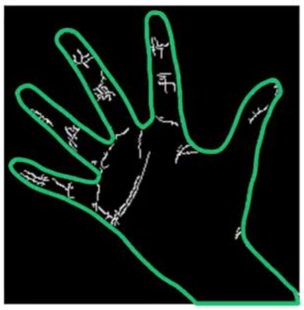
In OpenCv contours are best detected when there`s a white object against a black background so before can identify contours in an image we first to create a binary threshold that image that has black and white pixels that distinguish different objects in an image



We will then use the edges of these objects to form contours

Binary image

These binary image is often produced by a simple threshold as shown here or by canny edge detector



Binary image

Let`s go to do the simple example

Hand recognition an going computer vision challenge where it`s useful for example to recognize are interpret sign language and recognizing a variety of other gestures   so let`s perform image contouring on this hand

Convert to grayscale

Create a binary threshold image

retval , binary = cv2.threshold(gray , 255 , 255 , cv2.THRESH_BINARY_INV)

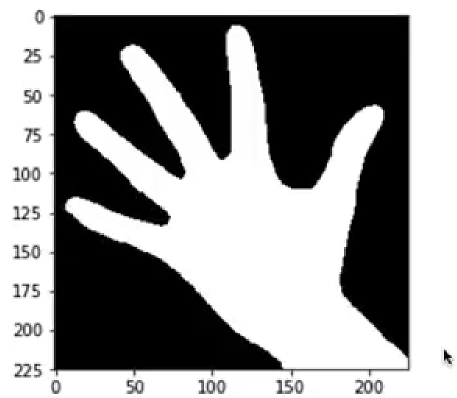
Display the plot

plt.imshow(binary , cmap = ‘ gray ‘)

I will use a binary threshold inverted to show the hand as way instead of the background to produce a binary image

Take a grayscale image in our grayscale image and isolates the white pixel values an it turn them black with inverse threshold

Then I will display this binary image



Find contours from threshold image

Function => cv2.findContours

Paramter -> binary image , contour retrieval mode which I will have a tree and third

is our approximation method which I will put as a simple chain and the outputs are list of contours in the hierarchy

The hierarchy is useful if you have many contours nested within one another

it is define the relationship one to another

you can learn more about this in text

retval , contours , hierarchy = cv2.findContours(binary , cv2.RETR_TREE , cv2.CHAIN_APPROX_SIMPLE)

Draw all contours on a copy the original image

The once we have a list of detected contours I will display them on a copy of our Image

Image_copy2 = np.copy(image_copy)

to draw the contours

Function drawContours()

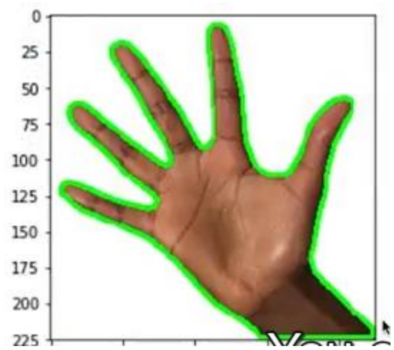
Paramter 1^{st}=> image_copy2 2^{nd}=> contours our list then which contours to display 3^{rd} => negative one means all the contours

4 => finally the color I want the contours to be have (0 , 255 , 0) then a green line and I display the output

5 => size I want the contours to be have 2

all_contours = cv2.drawContours(image_copy , contours , -1 , (0 , 255 , 0) , 2)

plt.imshow(all_contours)



You can see there`s one contour that nicely define the boundaries an outline of the hand

You can also see that it creates a complete closed boundary and this case it also separates the image into this foreground object and a background

I can extract lots of information about the shape of the hand including the

1- Area

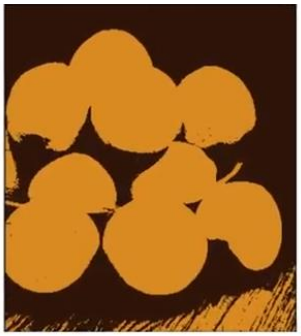
2- Center of the shape
3- The perimeter
4- Boundary Rectangle

K-means Cluster

One commonly used image segmentations technique is k-means clustering

It`s is machine learning technique that separate an image into segment by clustering or grouping together data points that have similar traits

An example let`s look at this image of oranges in a bowl , if I asked k-means to break up this image into two different colors it will give us an image that look like this



The oranges and the lighter part of table are all considered to be one cluster of orange points and the dark background and any part in shadow are clustered as dark brown data points.

But we `re ahead of ourselves

Let`s talk about k-means clusters this data step by step

k-means called unsupervised learning methods which means you don`t need to label data instead unsupervised learning aims to group and characterize unlabeled datasets

- identifies pattern and similarities in group of data so you can give k-means a set of any unlabeled data like the pixel values in an image and just tell it into break it into k clusters where k is a variable whose value choose k = number of cluster

For example we choose a k is 2 or 2 or 6

k = 2

k = 3

k = 6

We would get three or six-different colored segments in this case those segment divide differently shared parts of the orange and the table

Let`s go through a simple example in even more detail

IMAGE SEGEMENTATION

Example ast8for allah

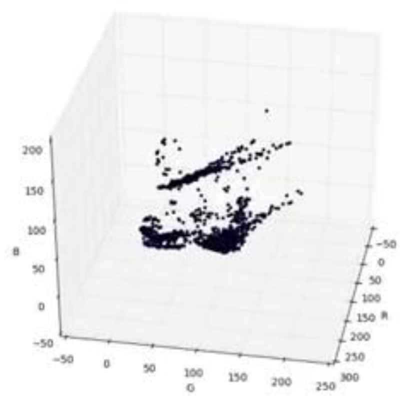
Rainbow

k = 3

Here`s a small image just 34 x 34 pixels a rainbow pattern I will use k-means to separate the image into three clusters based on color To start , we know that each pixel in this image has associated RGB value

F(x,y) = [R , G , B]

in Fact , we can actually plot the value of each pixel as a data point in RGB color space.



The axis are just values for R , G and B

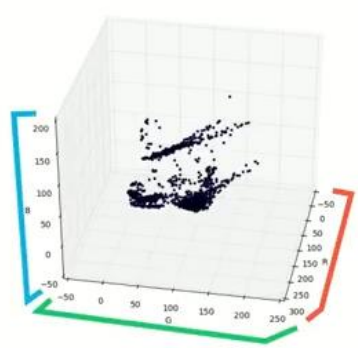
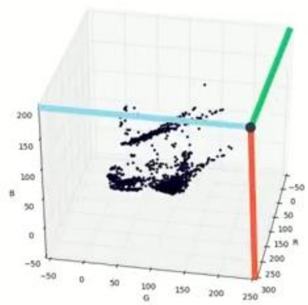
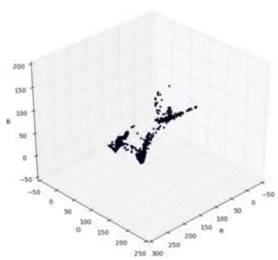


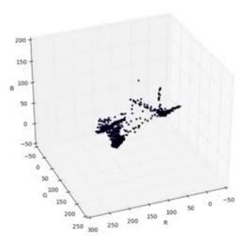
IMAGE SEGEMENTATION

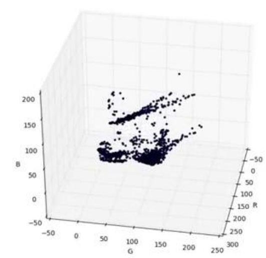
Here the highest R,G and B



We can actually see that the points fall into natural color clusters





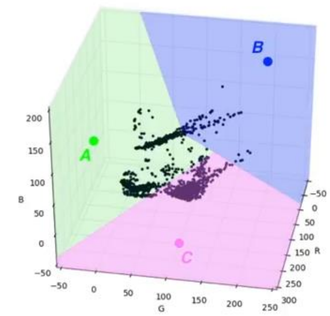


If I tell k-means to separate this image data into three cluster k = 3

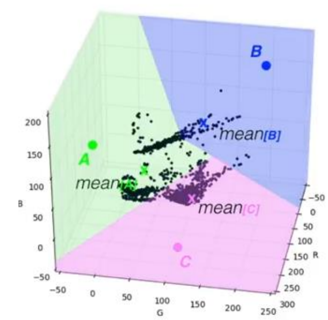
It will look at these pixel values and randomly guess three RGB points that partition the data into three clusters

1- Choose k random center points
   a. Center point A
   b. Center point B
   c. Center point C
2- Assign every data point to a cluster , based on its nearest center point so all this pixels on the left to cluster A and these more to the right cluster B and the bottom to cluster C and here I divided each into a separately colored region
3- Takes the mean (average) of the all values in each cluster of al RGB values in each cluster
   a. the mean values become the updated values for each center point ,

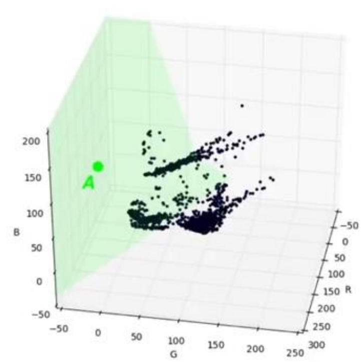
Step one and two



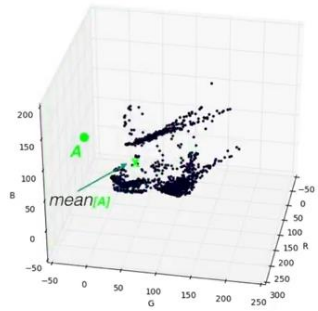
Step three



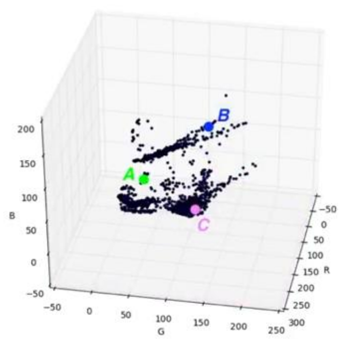
Let`s go back a step and take one cluster as A



k-means look at all the pixel values in this cluster and computes a mean RGB value let`s called mean a



And then updated the position of the center point A to be this mean value This is why I initially called these center points , They `re supposed to be in the center of their cluster And it does the same thing for cluster B and center point B and Cluster C and center point C



Moving the guessed center points to the mean of their cluster values Then this process repeats New Clusters are formed based on where they are in relation to theses new adjusted center points And then , an updated mean is calculated from these clusters and the center points are updated again So, there`s a sort of pulling done by the data points to create a new center points

4- The process repeat again 2 and 3 until convergence is reached (the center points will generally move a smaller and smaller distance from their previous value after each iteration The algorithm keeps repeating these steps until it converges and the convergence is defined by us it`s usually after a number of iterations say 10 or based on how much the center points move after each iteration)

Note

If the center points move less than some small values , say one pixel during an iteration then the center points have converged and the algorithm is finished

The Center points that arise at the end of this process should best separate the data