OpenCV

Introduction To OpenCV

1-How images are stored

Images in computer are stores in arrays where each array point represent the pixel of image and its value range from 0-255 for coloured images we use 3d array with depth three and each layer representing red, blue and green

---

2-Intro to open CV

To interact and compute on images we use openCV

```
import cv2
#to load image imread is used
image_bla = cv2.imread('address of image')
#to display image imshow is used
cv2.imshow('title of image', image_bla)
#waitkey allow us to input information when our image is open
cv2.waitkey()
#this close all the open window
cv2.destroyAllWindow()
import numpy as np
# return the shape of the image
image_bla.shape
# to save the image use imwrite
cv2.imwrite('output.png', image_bla)
```

3-Grayscaling

grayscaling is the process in which image is converted from full color to shades of grey

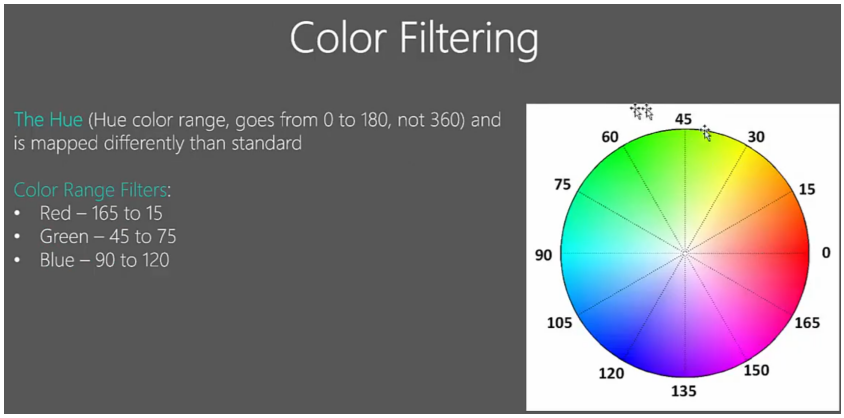
```
#to convert the image we use cvtcolor function
gray_image = cv2.cvtColor(image_bla, cv2.COLOR_BGR2GREY)
#this can be done easier also by passing argument 0 while loading the image
image_bla = cv2.imread('image address', 0)
```

4-Color spaces

color spaces are different in with we can save images eg - RBG, HSV, CMYK etc

will be stored as 0xRRGGBB00.

although we use rgb normally but for color filtering we use HSV because it way easier to filter color in hsv format




```
# to convert image to hsv
hsv_image = cv2.cvtColor(image_bla, cv2.COLOR_BGR2HSV)
#to split the image into different layer we use split function
r , b, g = cv2.split(image_bla)
# we can also merge these layers
merged = cv2.merge([b, g, r])
#to see the actual color
zero = np.zeros(image.shape[:2], dtype="uint8")
red_image = cv2.merge([zero,zero,r])
cv2.imshow('red', red_image)
```

5-Visualising Images as Histogram

It help us visualise which intensity of color is more in image

```
from matplotlib import pyplot as plt
image = cv2.imread('image_address')
histogram = cv2.calcHist([image], [0], None, [256], [0, 256])
plt.hist(image.ravel(), 256, [0, 256])
plt.show()
# Viewing separate Color Channels
```

```
color = ('b', 'g', 'r')
for i, col in enumerate(color):
     histogram2 = cv2.calcHist([image], [i], None, [256], [0, 256])
     plt.plot(histogram2, color = col)
     plt.xlim([0,256])
plt.show()
```

6-Drawing on Images

cv2.line() – This function is used to draw line on an image.

cv2.rectangle() – This function is used to draw rectangle on an image.

cv2.circle() – This function is used to draw circle on an image.

cv2.putText() – This function is used to write text on image.

cv2.ellipse() – This function is used to draw ellipse on image.

cv2.polylines — This function is used to draw ploygons

```
# creating black image of 350x350 just for demo
import numpy as np
import cv2
my_img = np.zeros((350, 350, 3), dtype = "uint8")
cv2.imshow('Window', my_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

1- Line

For drawing a line cv2.line() function is used. This function takes five arguments

- Image object on which to draw

- Starting point coordinates (x, y)

- End point coordinates (x, y)

- Stroke color in BGR (not RGB, to be noted)

- Stroke thickness (in pixels)

```
# creating for line
cv2.line(my_img, (202, 220), (100, 160), (0, 20, 200), 10)
cv2.imshow('Window', my_img)
```

2- Rectangle

For drawing a rectangle cv2.rectangle() function is used. This function accepts five input parameters.

- Image object on which to draw

- Coordinates of the vertex at the top left (x, y)

- Coordinates of the lower right vertex (x, y)

- Stroke color in BGR (not RGB, to be noted)

- Stroke thickness (in pixels)

```
# creating a rectangle
cv2.rectangle(my_img, (30, 30), (300, 200), (0, 20, 200), 10)
cv2.imshow('Window', my_img)
```

3-Circle

For drawing a circle, cv2.circle() function is used. This function accepts five input parameters.

- Image object on which to draw
- Center coordinates (x, y)
- Radius of the circle
- Stroke color in BGR (not RGB, to be noted)
- Stroke thickness (in pixels)

```
# creating circle
cv2.circle(my_img, (200, 200), 80, (0, 20, 200), 10)
cv2.imshow('Window', my_img)
```

4-Ellipse

For drawing an ellipse, cv2.ellipse() function is used. This function accepts eight input parameters.

- Image object on which image to be drawn
- Center coordinates (x, y)
- Length of the minor and major axes (h, w)
- Rotation angle of the ellipse (calculated counterclockwise)
- Starting angle (calculated clockwise)
- Final angle (calculated clockwise)
- Stroke color in BGR (not RGB to be noted)
- Stroke thickness

```
# creating ellipse
cv2.ellipse(my_img,(256,256),(100,50),0,0,180,255,-1)
cv2.imshow('Window', my_img)
```

5-Polynomial

For drawing a polygon, cv2.polylines() function is used. This function needs five number of arguments.

- The image object on which to draw
- The array of coordinates
- True, if it is a closed line
- Stroke color
- Stroke thickness

```
#creating Polynomial
pts = np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)
pts = pts.reshape((-1,1,2))
cv2.polylines(my_img,[pts],True,(0,255,255), 3)
cv2.imshow('Window', my_img)
```

6- Text

For text, cv2.putText() function is used.

- The image object on which to draw
- Text string to be drawn.
- coordinates of the bottom-left corner of the text string in the image
- font type. Some of font types are FONT_HERSHEY_SIMPLEX, FONT_HERSHEY_PLAIN, , etc
- Font scale factor
- Stroke color
- Stroke thickness

```
#creating Text
cv2.putText(image, 'Hello world!', (75, 290), cv2.FONT_HERSHEY_COMPLEX, 2, (100, 170, 0), 3)
cv2.imshow('Hello world', image)
```

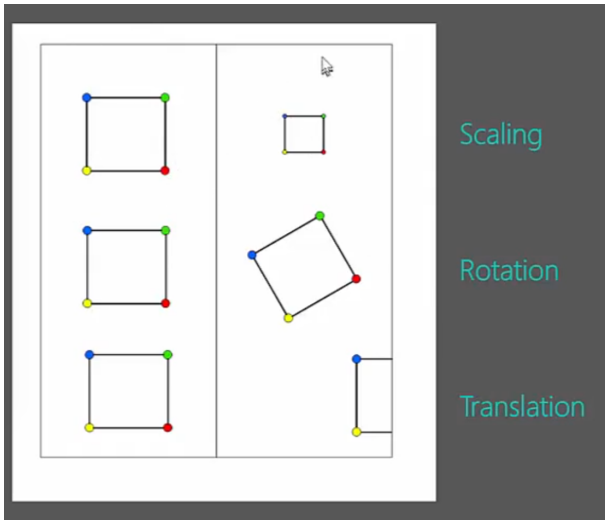
Image Manipulations

Transformations

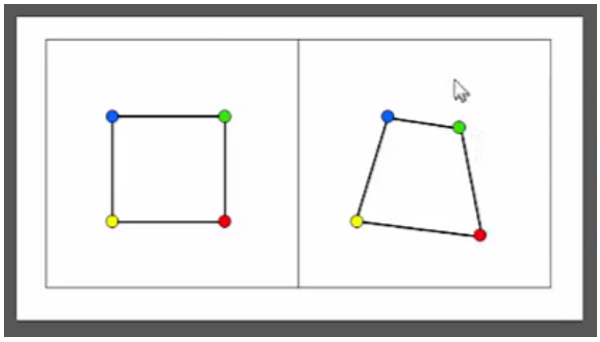
Transformations are geometric distorsions enacted upon an image. We use transformations to correct distortions or perspective issues from asising from the point of view an image was captured.

Types:

- Affine
- Scaling
- Rotation
- Translation



- Non- Affine or Projective Transform
- Non affine transformation does not preserve parallelism, length, and angle. It does however preserve collinearity and incidence.



---

1 - Translations

This is an affine transformation that simply shift the image in x and y direction

cv2.warpAffine function is used for translation

Translation matrix

T = [1 0 Tx

0 1 Ty]

Tx — shift along x axis

Ty — shift along y axis

```
# translation
T = np.float32([[1, 0, Tx], [0, 1, Ty]])
t_img = cv2.warpAffine(image, T, (width, height ))
cv2.imshow('Translated Image', t_img)
```

---

2 - Rotations

This simply rotate the image

Rotation matrix

R = [cosØ -sinØ

sinØ cosØ]

Ø — angle of rotation

cv2.getRotationMatrix2D function is used to create rotation matrix

cv2.warpAffine funtion is used for rotation

```
#rotation
R = cv2.getRotationMatrix2D((width/2, height/2), Ø, 1)
r_img = cv2.warpAffine(image, R, (wigth, height))
cv2.imshow('Rotated image', r_img)
```

---

3 - Scaling

Scaling is very important image transformation. Image scaling can reduce number of pixels thus reduce training time for a model.

Scaling may seem simple on paper but math behind it is very complex as in scaling we have to recreate or merge pixels. OpenCV provide many interpolation methods for scaling.

Interpolation is a method of constructing new data points with the range of a discrete set of known data points

Choices for interpolation :

- cv2.INTER_AREA - good for shrinking images
- cv2.INTER_NEAREST - fastest
- cv2.INTER_LINEAR - good for zooming and default
- cv2.INTER_CUBIC - better
- cv2.INTER_LANCZOS4 - best

cv2.resize function is used for scaling

```
# Let's make our image 3/4 of it's original size
image_scaled = cv2.resize(image, None, fx=0.75, fy=0.75)
```

```
cv2.imshow('Scaling - Linear Interpolation', image_scaled)
# Let's double the size of our image
img_scaled = cv2.resize(image, None, fx=2, fy=2, interpolation = cv2.INTER_CUBIC)
cv2.imshow('Scaling - Cubic Interpolation', img_scaled)
# Let's skew the re-sizing by setting exact dimensions
img_scaled = cv2.resize(image, (900, 400), interpolation = cv2.INTER_AREA)
cv2.imshow('Scaling - Skewed Size', img_scaled)
```

Note - to download images directly from internet and use them

```
import urllib
def url_to_image(url):
     # download the image, convert it to a NumPy array, and then read
     # it into OpenCV format
     resp = urllib.request.urlopen(url)
     image = np.asarray(bytearray(resp.read()), dtype="uint8")
     image = cv2.imdecode(image, cv2.IMREAD_COLOR)
     # return the image
     return image
image = url_to_image('url')
```

Image Pyramiding

There is another Technique for image scaling called image pyramiding. What it does is shrinking and enlarging multiple copies of a image, this technique is quite useful in object detection as it create multiple copies of a image with diff scaling.

cv2.pyrDown create half the size of orignal image

cv2.pyrUp create twice the size of orignal image

```
smaller = cv2.pyrDown(image)
larger = cv2.pyrUp(smaller)
cv2.imshow('Smaller ', smaller )
cv2.imshow('Larger ', larger )
```

---

4 - Cropping

Cropping is extracting an segment of image. It is fairly staight forward as image is represent in form of numpy array, we simply use indexing to crop desired numpy array

```
# cropping
# getting height and width of image
height, width = image.shape[:2]
# Let's get the starting pixel coordiantes (top  left of cropping rectangle)
start_row, start_col = int(height * .25), int(width * .25)
# Let's get the ending pixel coordinates (bottom right)
end_row, end_col = int(height * .75), int(width * .75)
# Simply use indexing to crop out the rectangle we desire
cropped = image[start_row:end_row , start_col:end_col]
cv2.imshow("Cropped Image", cropped)
```

---

5 - Arithmetic Operations

We use arithmetic operation on the image to light and darken the image. We normally took matrix of 75 and by subtracting we darken the image and by adding we lighten the image

```
# creating matrix of 75 of same dimension
M = np.ones(image.size, dtype="uint8")
#we add m to image to increase brightness
added = cv2.add(image, M)
cv2.imshow("added", added)
#we subtract m to image to decrease the brightness
subtracted = cv2.subtract(image, M)
cv2.imshow("subtracted", subtracted)
```

---

6 - Bitwise Operations

We can perform bitwise operation on images like and, or, nor, not.

```
#show only intersection
and = cv2.bitwise_and(image1, image2)
cv2.imshow("And", and)
#shows union
or = cv2.bitwise_or(image1, image2)
cv2.imshow("Or", or)
#show where either exists
nor = cv2.bitwise_nor(image1, image2)
cv2.imshow("Nor", nor)
#show everything that isn't part of image
not = cv2.bitwise_not(image1)
cv2.imshow("Not", not)
```

---

7 - Convolution

In image processing, a **kernel**, **convolution matrix**, or **mask** is a small matrix. It is used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between a kernel and an image.

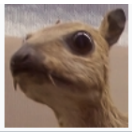
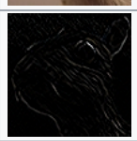
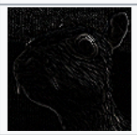
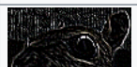
convolution is a mathematical operator which takes two functions *f* and *g* and produces a third function.

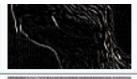
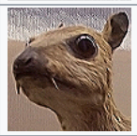
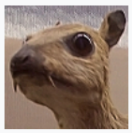
The general expression of a convolution is

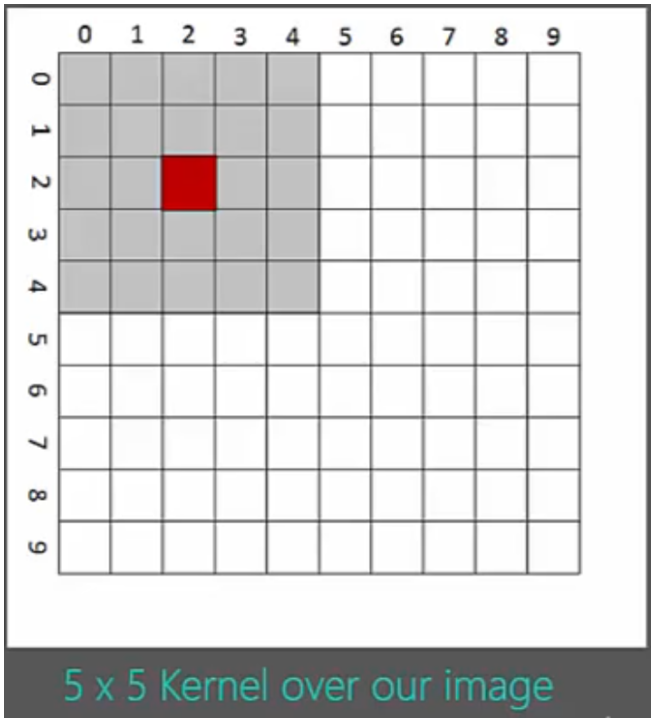
$g(x, y) = \omega * f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} \omega(s, t) * f(x-s, y-t)$

where g(x, y) is the filtered image, f(x, y) is the original image, ω is the filter kernel.

Different kernels

Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \end{bmatrix}$	

	$\begin{bmatrix} -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	
Unsharp masking 5 × 5 Based on Gaussian blur with amount as 1 and threshold as 0 (with no image mask)	$\frac{-1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & -476 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	



cv2.filter2D(image, -1, kernel) function is used for applying kernel filters to image

1- Blurring

Using blur kernel

```
#Creating 3x3 kernel of 1
blur_kernel = np.ones((3, 3), np.float32) / 9
blurred = cv2.filter2D(image, -1, blur_kernel)
```

```
cv2.imshow(“blurred image”, blurred)
```

Using box filter

```
#Box filter average image pixel by convolving with box filter
 #Box size need to be odd and positive
 blur = cv2.blur(image, (3, 3))
 cv2.imshow(“blurred image”, blur)
```

Using Gaussian filter

```
#Gaussian kernel is same as box filter but its kernal is gaussian in nature
 G_blur = cv2.GaussianBlur(image, (7, 7), 0)
 cv2.imshow(“blurred image”, G_blur)
```

Using Median filter

```
#Median the values of the kernel and it added a painted type of effect
 m_blur = cv2.medianBlur(image, 5)
 cv2.imshow(“blurred image”, m_blur)
```

Using Bilateral filter

```
#Bilateral filter preserve vertical and horizontal edges of the image
 b_blur = cv2.bilateralFilter(image, 9, 75, 75)
 cv2.imshow(“blurred image”, b_blur)
```

Image De-noising - Non-Local Means Denoising

There are 4 variations of Non-Local Means Denoising:

- cv2.fastNlMeansDenoising() - works with a single grayscale images

- cv2.fastNlMeansDenoisingColored() - works with a color image.

- cv2.fastNlMeansDenoisingMulti() - works with grayscale images captured in short period of time

- cv2.fastNlMeansDenoisingColoredMulti() - same as above, but for color images.

```
# Parameters, after None are - the filter strength 'h' (5-10 is a good range)
# Next is hForColorComponents, set as same value as h again
dst = cv2.fastNlMeansDenoisingColored(image, None, 6, 6, 7, 21)
cv2.imshow('Fast Means Denoising', dst)
```

2 - Sharpening

```
# Create our shapening kernel, we don't normalize since the
# the values in the matrix sum to 1
kernel_sharpening = np.array([[-1,-1,-1],
                              [-1,9,-1],
                              [-1,-1,-1]])
# applying different kernels to the input image
sharpened = cv2.filter2D(image, -1, kernel_sharpening)
cv2.imshow('Image Sharpening', sharpened)
```

8 - Thresholding

Thresholding is act of converting an image into binary form. All threshold function uses grayscale images, images need to be converted into grayscale first.

cv2.threshold(image, Threshold value, Max value, Threshold Type) function is used for thresholding

Threshold types :

- cv2.THRESH_BINARY - Most common
- cv2.THRESH_BINARY_INV
- cv2.THRESH_TRUNC
- cv2.THRESH_TOZERO
- cv2.THRESH_TOZERO_INV

```
# Values below 127 goes to 0 (black, everything above goes to 255 (white)
ret,thresh1 = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY)
cv2.imshow('1 Threshold Binary', thresh1)


# Values below 127 go to 255 and values above 127 go to 0 (reverse of above)
ret,thresh2 = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY_INV)
cv2.imshow('2 Threshold Binary Inverse', thresh2)


# Values above 127 are truncated (held) at 127 (the 255 argument is unused)
ret,thresh3 = cv2.threshold(image, 127, 255, cv2.THRESH_TRUNC)
cv2.imshow('3 THRESH TRUNC', thresh3)


# Values below 127 go to 0, above 127 are unchanged
ret,thresh4 = cv2.threshold(image, 127, 255, cv2.THRESH_TOZERO)
cv2.imshow('4 THRESH TOZERO', thresh4)


# Resever of above, below 127 is unchanged, above 127 goes to 0
ret,thresh5 = cv2.threshold(image, 127, 255, cv2.THRESH_TOZERO_INV)
cv2.imshow('5 THRESH TOZERO INV', thresh5)
```

Is there a better way off thresholding?

The biggest downfall of those simple threshold methods is that we need to provide the threshold value (i.e. the 127 value we used previously).

What if there was a smarter way of doing this?, There is with, Adaptive thresholding. There are two method for adaptive thresholding

- Adaptive mean thresholding
- Otsu's thresholding

cv2.aptiveThreshold(image, Max value, Adaptive type, Threshold type, Block size, Constant that is subtracted from mean)

NOTE: block size need to be odd number

Adaptive threshold type :

- ADAPTIVE_THRESH_MEAN_C

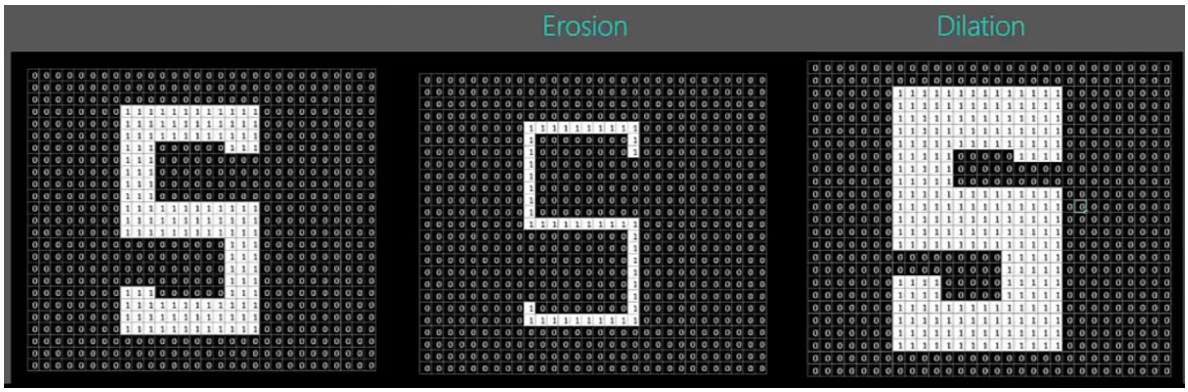
- ADAPTIVE THRESH_GAUSSIAN_C

- THRESH_OTSU - best method it looks for two peak value in histogram of image and find optimal value that separate these two peak best

```
# Using adaptiveThreshold
thresh = cv2.adaptiveThreshold(image, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 3, 5)
cv2.imshow("Adaptive Mean Thresholding", thresh)
th2 = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
cv2.imshow("Otsu's Thresholding", th2)
```

---

9 - Dilation and Erosion



Dilation - add pixels to boundaries of objects in an image

Erosion - removes pixels at the boundaries of object in an image

Opening - Erosion followed by dilation

Closing - Dilation followed by Erosion

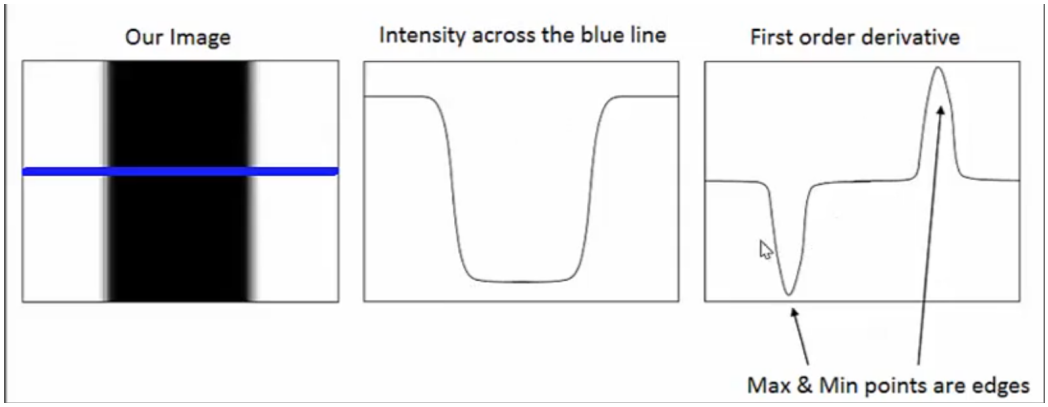
NOTE: Sometime we feel Dilation and Erosion have opposite affect because it consider white as the object

```
# Let's define our kernel size
kernel = np.ones((5,5), np.uint8)
# Now we erode
erosion = cv2.erode(image, kernel, iterations = 1)
cv2.imshow('Erosion', erosion)
# Now we dilate
dilation = cv2.dilate(image, kernel, iterations = 1)
cv2.imshow('Dilation', dilation)
# Opening - Good for removing noise
opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
cv2.imshow('Opening', opening)
# Closing - Good for removing noise
closing = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
cv2.imshow('Closing', closing)
```

---

10 - Edge Detection

Edge detection is very important in computer vision. Edge can be define as sudden change in an image and they encode as much info as pixels



There are three type of edge Detection :

- Sobel - to emphasize vertical or horizontal edges

- Laplacian - Gets all orientations

- Canny - Optimal due to low error rate, well define edge and accurate detection

1 - Sobel

cv2.Sobel(original_image,ddepth,xorder,yorder,kernelsize)

```
# Extract Sobel Edges
sobel_x = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=5)
sobel_y = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=5)
cv2.imshow('Sobel X', sobel_x)
cv2.imshow('Sobel Y', sobel_y)
sobel_OR = cv2.bitwise_or(sobel_x, sobel_y)
cv2.imshow('sobel_OR', sobel_OR)
```

2 - Laplacian

```
laplacian = cv2.Laplacian(image, cv2.CV_64F)
cv2.imshow('Laplacian', laplacian)
```

3 - Canny Edge detection is most widely used algo

- Applies gaussian blurring to reduce noise

- Finds intensity gradient of the image - Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to find edge gradient and direction for each pixel. Gradient direction is always perpendicular to edges.

- Applied non-maximum suppression(i.e. removes pixels that are not edge)

- Hysteresis - Applies threshold (i.e. if pixel is within the upper threshold, it is considered an edge)

cv2.Canny(image, threshold 1, threshold 2) function is used for canny edge detection Any gradient value larger than threshold2 is considered to be an edge. Any value below threshold1 is considered not to be an edge.

```
#Canny edge detection
canny = cv2.Canny(image, 50, 120)
cv2.imshow('Canny', canny)
```

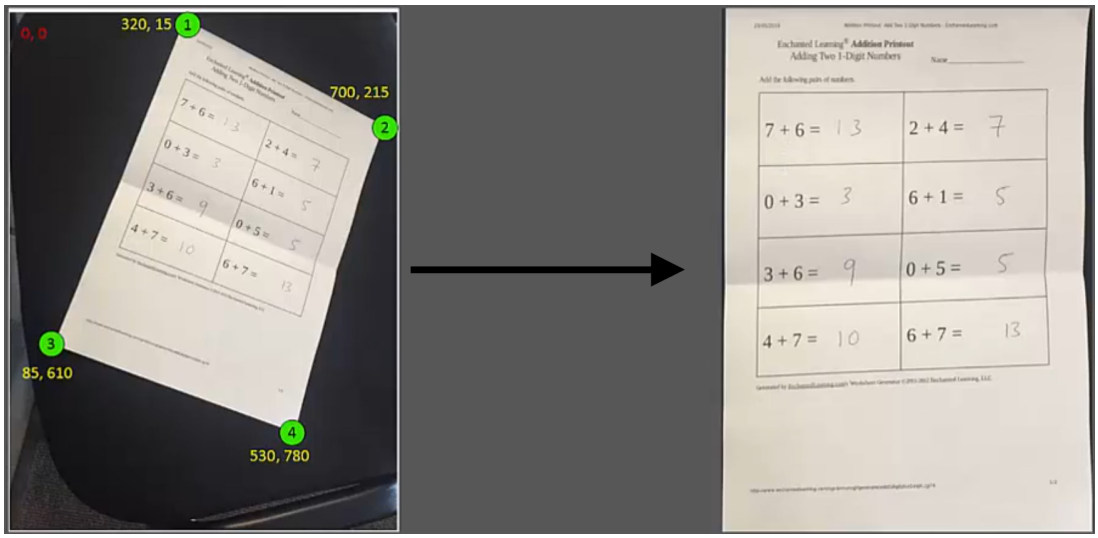
---

11 - Perspective Transformation

When an image shot at diff angle other than perpendicular to object we get angled perspective.

Non-Affine Transform

Non-affine transformation is used to flattend that image and get a clear view of the object. It change the perspective of image




```
# Cordinates of the 4 corners of the original image
points_A = np.float32([[320,15], [700,215], [85,610], [530,780]])
# Cordinates of the 4 corners of the desired output
# We use a ratio of an A4 Paper 1 : 1.41
points_B = np.float32([[0,0], [420,0], [0,594], [420,594]])
# Use the two sets of four points to compute
# the Perspective Transformation matrix, M
M = cv2.getPerspectiveTransform(points_A, points_B)
warped = cv2.warpPerspective(image, M, (420,594))
cv2.imshow('warpPerspective', warped)
```

Affine Transform

If we use three points instead of four and thats called affine transform, and the image will be rotated but perspective will not change means lines will remain parallel which are in orignal image

```
# Cordinates of the 4 corners of the original image
points_A = np.float32([[320,15], [700,215], [85,610]])
# Cordinates of the 4 corners of the desired output
# We use a ratio of an A4 Paper 1 : 1.41
points_B = np.float32([[0,0], [420,0], [0,594]])
# Use the two sets of four points to compute
# the Perspective Transformation matrix, M
M = cv2.getAffineTransform(points_A, points_B)
warped = cv2.warpAffine(image, M, (cols, rows))
cv2.imshow('warpPerspective', warped)
```

---

12 - Capturing Video from web cam

cap = cv2.VideoCapture(0) - function is used to active webcam

ret, frame = cap.read() - function capture frames from webcam

cv2.imshow('Our Live Sketcher', frame) - will display the output frame

we want to run cap.read to run in a loop so it can capture frames continuesly so it look like video

```
# Initialize webcam, cap is the object provided by VideoCapture
# It contains a boolean indicating if it was sucessful (ret)
# It also contains the images collected from the webcam (frame)
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    #we can apply some cv operation on the frames
    cv2.imshow('Our Live Sketcher', frame)
    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break

# Release camera and close windows
cap.release()
cv2.destroyAllWindows()
```

Image Segmentation Contours

Image segmentation is process of partitioning images into diff partitions

Contours are continuous lines or curves that bound or cover the full boundary of an object in an image. Contours are very important in object detection and shape analysis. While finding contours we first convert it to greyscale then we can find canny edges it is not required but help in avoiding unwanted contours.

Basically contours are indivisual edges, we apply contours so we can access detected edges individually.

cv2.findContours(image, Retrieval Mode, Approximation Method)

Returns -> ret, contours, hierarchy

The variable 'contours' are stored as a numpy array of (x,y) points that form the contour

Approximation Methods

- cv2.CHAIN_APPROX_NONE stores all the boundary points. But we don't necessarily need all bounding points. If the points form a straight line, we only need the start and ending points of that line.

- cv2.CHAIN_APPROX_SIMPLE instead only provides these start and end points of bounding contours, thus resulting in much more efficent storage of contour information

Retrival Mode

- cv2.RETR_LIST - Retrieves all contours

- cv2.RETR_EXTERNAL - Retrieves external or outer contours only - most useful

- cv2.RETR_COMP - Retrieves all in a 2-level hierarchy

- cv2.RETR_TREE - Retrieves all in full hierarchy

```
# Grayscale
```

```
gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
# Find Canny edges
edged = cv2.Canny(gray, 30, 200)
# Finding Contours
# Use a copy of your image e.g. edged.copy() if u don't want your image to alter, since
findContours alters the image
contours, hierarchy = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
cv2.imshow('Canny Edges After Contouring', edged)
print("Number of Contours found = " + str(len(contours)))
# Draw all contours
# Use '-1' as the 3rd parameter to draw all
# U can also pass index to draw particular contour eg - if u pass 1 it will draw first contour
cv2.drawContours(image, contours, -1, (0,255,0), 3)
cv2.imshow('Contours', image)
```

---

1 - Sorting Contours

- sorting contours by Area can eliminate small contours that may be noise
- sorting by spatial postion sort characters left to right

sorting contours by Area :

```
def get_contour_areas(contours):
    # returns the areas of all contours as list
    all_areas = []
    for cnt in contours:
        area = cv2.contourArea(cnt)
        all_areas.append(area)
    return all_areas
# Let's print the areas of the contours before sorting
print("Contor Areas before sorting")
print(get_contour_areas(contours))
# Sort contours large to small
sorted_contours = sorted(contours, key=cv2.contourArea, reverse=True)
#sorted_contours = sorted(contours, key=cv2.contourArea, reverse=True)[:3]
print("Contor Areas after sorting")
print(get_contour_areas(sorted_contours))
# Iterate over our contours and draw one at a time
for c in sorted_contours:
    cv2.drawContours(orginal_image, [c], -1, (255,0,0), 3)
    cv2.imshow('Contours by area', orginal_image)
```

sorting contours by Spatial position :

I Think it is useless - sorting by spatial position exept may be used for numbering or naming them

```
# Functions we'll use for sorting by position
def x_cord_contour(contours):
    #Returns the X cordinate for the contour centroid
    if cv2.contourArea(contours) > 10:
        M = cv2.moments(contours)
        return (int(M['m10']/M['m00']))
    else:
        pass
def label_contour_center(image, c):
```

```
    # Places a red circle on the centers of contours
    M = cv2.moments(c)
    cx = int(M['m10'] / M['m00'])
    cy = int(M['m01'] / M['m00'])
    # Draw the countour number on the image
    cv2.circle(image,(cx,cy), 10, (0,0,255), -1)
    return image
# Load our image
image = cv2.imread('images/bunchofshapes.jpg')
orginal_image = image.copy()
# Computer Center of Mass or centroids and draw them on our image
for (i, c) in enumerate(contours):
    orig = label_contour_center(image, c)
cv2.imshow("4 - Contour Centers ", image)
cv2.waitKey(0)
# Sort by left to right using our x_cord_contour function
contours_left_to_right = sorted(contours, key = x_cord_contour, reverse = False)
# Labeling Contours left to right
for (i,c)  in enumerate(contours_left_to_right):
    cv2.drawContours(orginal_image, [c], -1, (0,0,255), 3)
    M = cv2.moments(c)
    cx = int(M['m10'] / M['m00'])
    cy = int(M['m01'] / M['m00'])
    cv2.putText(orginal_image, str(i+1), (cx, cy), cv2.FONT_HERSHEY_SIMPLEX, 2, (0, 255, 0), 3)
    cv2.imshow('6 - Left to Right Contour', orginal_image)
    cv2.waitKey(0)
    (x, y, w, h) = cv2.boundingRect(c)

    # Let's now crop each contour and save these images
    cropped_contour = orginal_image[y:y + h, x:x + w]
    image_name = "output_shape_number_" + str(i+1) + ".jpg"
    print(image_name)
    cv2.imwrite(image_name, cropped_contour)
```

---

2 - Approximating Contours

cv2.approxPolyDP(contour, Approximation Accuracy, Closed)

- contour – is the individual contour we wish to approximate

- Approximation Accuracy – Important parameter is determining the accuracy of the approximation. Small values give precise-approximations, large values give more generic approximation. A good rule of thumb is less than 5% of the contour perimeter

- Closed – a Boolean value that states whether the approximate contour should be open or closed

```
# Find contours
contours, hierarchy = cv2.findContours(thresh.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)
# Iterate through each contour and compute the bounding rectangle
for c in contours:
    x,y,w,h = cv2.boundingRect(c)
    cv2.rectangle(orig_image,(x,y),(x+w,y+h),(0,0,255),2)
    cv2.imshow('Bounding Rectangle', orig_image)
cv2.waitKey(0)

# Iterate through each contour and compute the approx contour
for c in contours:
```

```
for c in contours:
    # Calculate accuracy as a percent of the contour perimeter
    accuracy = 0.03 * cv2.arcLength(c, True)
    approx = cv2.approxPolyDP(c, accuracy, True)
    cv2.drawContours(image, [approx], 0, (0, 255, 0), 2)
    cv2.imshow('Approx Poly DP', image)
```

Convex Hull

```
# Sort Contors by area and then remove the largest frame contour to get rid of box on entire
image
n = len(contours) - 1
contours = sorted(contours, key=cv2.contourArea, reverse=False)[:n]
# Iterate through contours and draw the convex hull
for c in contours:
    hull = cv2.convexHull(c)
    cv2.drawContours(image, [hull], 0, (0, 255, 0), 2)
    cv2.imshow('Convex Hull', image)
```

---

3 - Matching Contours Shapes

Used to match shape in image eg: if we want to find shape similar in two images

cv2.matchShapes(contour template, contour, method, method parameter)

Output – match value (lower values means a closer match)

- Contour Template – This is our reference contour that we're trying to find in the new image
- Contour – The individual contour we are checking against
- Method – Type of contour matching (1, 2, 3)
- Method Parameter – leave alone as 0.0 (not fully utilized in python OpenCV)

Link to explore Method type of contour matching

http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html

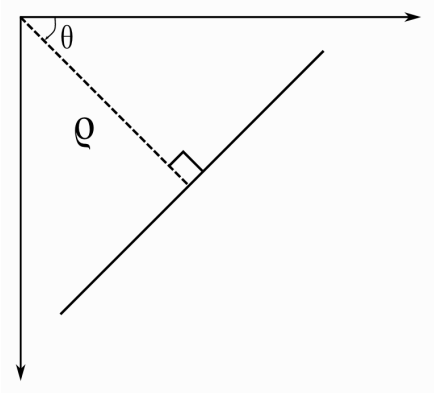
```
# We extract the contour we want in first image which will be our template contour
template_contour = contours[1]
# Extract contours from second image
contours, hierarchy = cv2.findContours(thresh2, cv2.RETR_CCOMP, cv2.CHAIN_APPROX_SIMPLE)
for c in contours:
    # Iterate through each contour in the second image and
    # use cv2.matchShapes to compare contour shapes
    match = cv2.matchShapes(template_contour, c, 3, 0.0)
    print(match)
    # If the match value is less than 0.15 we
    if match < 0.15:
        closest_contour = c
    else:
        closest_contour = []

cv2.drawContours(target, [closest_contour], -1, (0,255,0), 3)
cv2.imshow('Output', target)
```

4 - Line Detection

Hough Line Transform

Hough Transform is a popular technique to detect any shape, if you can represent that shape in mathematical form. It can detect the shape even if it is broken or distorted a little bit.

A line can be represented as y = mx+c or in parametric form, as p = x*cosø + y*sinø where p is the perpendicular distance from origin to the line, and ø is the angle formed by this perpendicular line and horizontal axis measured in counter-clockwise (That direction varies on how you represent the coordinate system. This representation is used in OpenCV).



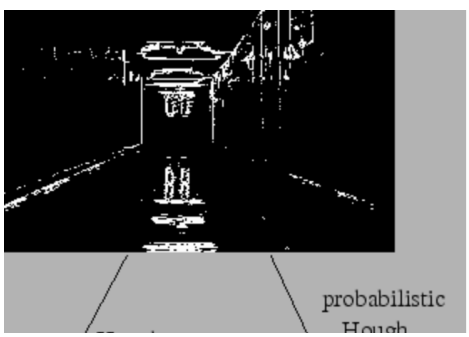
Apply greyscale and threshold or use canny edge detection before finding applying hough transform.

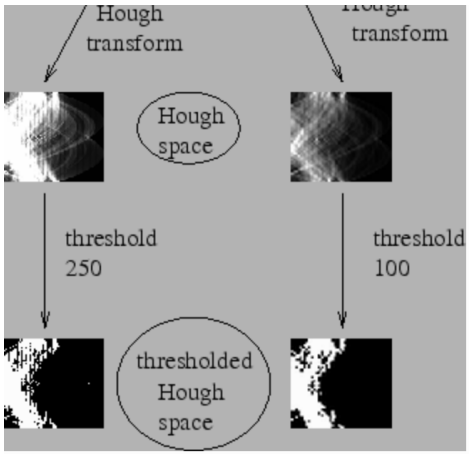
cv2.HoughLines(image, p accuracy, ø accuracy, threshold) function is used to detect line

```
lines = cv2.HoughLines(edges,1,np.pi/180,200)
for rho,theta in lines[0]:
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    cv2.line(img,(x1,y1),(x2,y2),(0,0,255),2)
cv2.imwrite('houghlines3.jpg',img)
```

Probabilistic Hough Transform

Probabilistic Hough Transform is an optimization of Hough Transform. It doesn't take all the points into consideration, instead take only a random subset of points and that is sufficient for line detection. Just we have to decrease the threshold.





The function used is cv2.HoughLinesP(). It has two new arguments.

- minLineLength - Minimum length of line. Line segments shorter than this are rejected.

- maxLineGap - Maximum allowed gap between line segments to treat them as single line.

```
img = cv2.imread('dave.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray,50,150,apertureSize = 3)
minLineLength = 100
maxLineGap = 10
lines = cv2.HoughLinesP(edges,1,np.pi/180,100,minLineLength,maxLineGap)
for x1,y1,x2,y2 in lines[0]:
    cv2.line(img,(x1,y1),(x2,y2),(0,255,0),2)
cv2.imwrite('houghlines5.jpg',img)
```

---

5 - Circle Detection

cv2.HoughCircles(image, method, dp, MinDist, param1, param2, minRadius, MaxRadius)

- Method - currently only cv2.HOUGH_GRADIENT available

- dp - Inverse ratio of accumulator resolution

- MinDist - the minimum distance between the center of detected circles

- param1 - Gradient value used in the edge detection

- param2 - Accumulator threshold for the HOUGH_GRADIENT method, lower allows more circles to be detected (false positives)

- minRadius - limits the smallest circle to this size (via radius)

- MaxRadius - similarly sets the limit for the largest circles

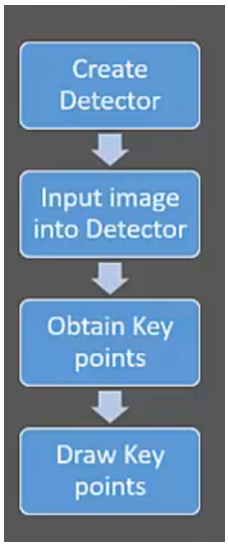
```
import cv2
import numpy as np
import cv2.cv as cv
image = cv2.imread('images/bottlecaps.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blur = cv2.medianBlur(gray, 5)
circles = cv2.HoughCircles(blur, cv.CV_HOUGH_GRADIENT, 1.5, 10)
for i in circles[0,:]:
       # draw the outer circle
       cv2.circle(image,(i[0], i[1]), i[2], (255, 0, 0), 2)

       # draw the center of the circle
```

```
        # draw the center of the circle
        cv2.circle(image, (i[0], i[1]), 2, (0, 255, 0), 5)
cv2.imshow('detected circles', image)
```

---

6 - Blob Detection

Blobs can be described as groups of connected pixels that all share a common property



The function cv2.drawKeypoints takes the following arguments:

cv2.drawKeypoints(input image, keypoints, blank_output_array, color, flags)

flags:

- cv2.DRAW_MATCHES_FLAGS_DEFAULT
- cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
- cv2.DRAW_MATCHES_FLAGS_DRAW_OVER_OUTIMG
- cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS

```
# Set up the detector with default parameters.
detector = cv2.SimpleBlobDetector_create()
# Detect blobs.
keypoints = detector.detect(image)
# Draw detected blobs as red circles.
# cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures the size of
# the circle corresponds to the size of blob
blank = np.zeros((1,1))
blobs = cv2.drawKeypoints(image, keypoints, blank, (255,0,0), cv2.DRAW_MATCHES_FLAGS_DEFAULT)
number_of_blobs = len(keypoints)
text = "Total Number of Blobs: " + str(len(keypoints))
cv2.putText(blobs, text, (20, 550), cv2.FONT_HERSHEY_SIMPLEX, 1, (100, 0, 255), 2)
# Show keypoints
cv2.imshow("Blobs", blobs)
```

We can add parameter to our blob detector algo to filter certain type of blob

parameters are :

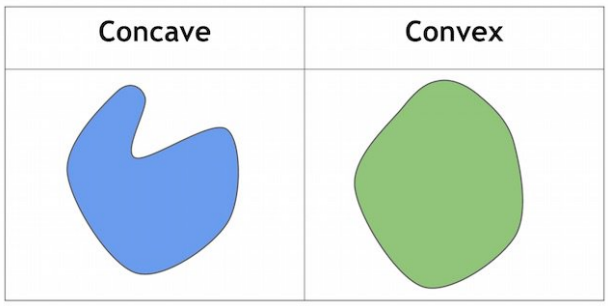
- Thresholding : Convert the source images to several binary images by thresholding the source image with thresholds starting

- Thresholding : Convert the source images to several binary images by thresholding the source image with thresholds starting at minThreshold. These thresholds are incremented by thresholdStep until maxThreshold. So the first threshold is minThreshold, the second is minThreshold + thresholdStep, the third is minThreshold + 2 x thresholdStep, and so on.
- Grouping : In each binary image, connected white pixels are grouped together. Let's call these binary blobs.
- Merging : The centers of the binary blobs in the binary images are computed, and blobs located closer than minDistBetweenBlobs are merged.
- Center & Radius Calculation : The centers and radii of the new merged blobs are computed and returned.

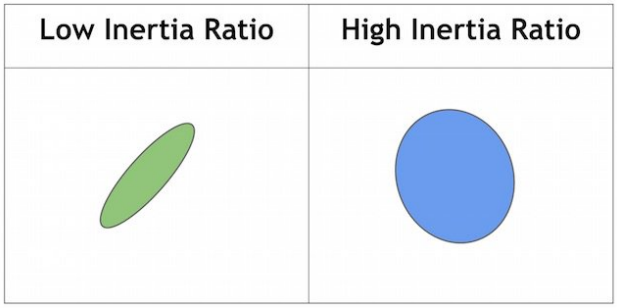
Filtering Blobs by Color, Size and Shape

The parameters for SimpleBlobDetector can be set to filter the type of blobs we want.

- By Color : [Note : This feature appears to be broken. I checked the code, and it appears to have a logical error] First you need to set filterByColor = 1. Set blobColor = 0 to select darker blobs, and blobColor = 255 for lighter blobs.
- By Size : You can filter the blobs based on size by setting the parameters filterByArea = 1, and appropriate values for minArea and maxArea. E.g. setting minArea = 100 will filter out all the blobs that have less then 100 pixels.
- By Shape : Now shape has three different parameters.
- Circularity : This just measures how close to a circle the blob is. E.g. a regular hexagon has higher circularity than say a square. To filter by circularity, set filterByCircularity = 1. Then set appropriate values for minCircularity and maxCircularity. This means that a circle has a circularity of 1, circularity of a square is 0.785, and so on.
- Convexity : A picture is worth a thousand words. Convexity is defined as the (Area of the Blob / Area of it's convex hull). Now, Convex Hull of a shape is the tightest convex shape that completely encloses the shape. To filter by convexity, set filterByConvexity = 1, followed by setting 0 ≤ minConvexity ≤ 1 and maxConvexity (≤ 1)



- Inertia Ratio : Don't let this scare you. Mathematicians often use confusing words to describe something very simple. All you have to know is that this measures how elongated a shape is. E.g. for a circle, this value is 1, for an ellipse it is between 0 and 1, and for a line it is 0. To filter by inertia ratio, set filterByInertia = 1, and set 0 ≤ minInertiaRatio ≤ 1 and maxInertiaRatio (≤ 1) appropriately.




```
# Set our filtering parameters
```

```
# Initialize parameter settiing using cv2.SimpleBlobDetector
params = cv2.SimpleBlobDetector_Params()
# Set Area filtering parameters
params.filterByArea = True
params.minArea = 100
# Set Circularity filtering parameters
params.filterByCircularity = True
params.minCircularity = 0.9
# Set Convexity filtering parameters
params.filterByConvexity = False
params.minConvexity = 0.2

# Set inertia filtering parameters
params.filterByInertia = True
params.minInertiaRatio = 0.01
# Create a detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)

# Detect blobs
keypoints = detector.detect(image)
# Draw blobs on our image as red circles
blank = np.zeros((1,1))
blobs = cv2.drawKeypoints(image, keypoints, blank, (0,255,0),
                                        cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
number_of_blobs = len(keypoints)
text = "Number of Circular Blobs: " + str(len(keypoints))
cv2.putText(blobs, text, (20, 550), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 100, 255), 2)
# Show blobs
cv2.imshow("Filtering Circular Blobs Only", blobs)
```

Object Detection

object detection vs recognition - recognation is second phase after object detection where it recognise the object thats detected.

First algorithm we gonna learning in object detection is template matching where we slides the template image our the source image untill the match is found

There are a variety of methods to perform template matching, but in this case we are using the correlation coefficient which is specified by the flag **cv2.TM_CCOEFF.**

So what exactly is the cv2.matchTemplate function doing? Essentially, this function takes a “sliding window” of our waldo query image and slides it across our puzzle image from left to right and top to bottom, one pixel at a time. Then, for each of these locations, we compute the correlation coefficient to determine how “good” or “bad” the match is.

Regions with sufficiently high correlation can be considered “matches” for our waldo template. From there, all we need is a call to cv2.minMaxLoc on Line 22 to find where our “good” matches are. That’s really all there is to template matching!

```
#finding waldo using template matching algo
# Load Template image
template = cv2.imread('./images/waldo.jpg',0)
result = cv2.matchTemplate(gray, template, cv2.TM_CCOEFF)
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)
#Create Bounding Box
```

```
top_left = max_loc
bottom_right = (top_left[0] + 50, top_left[1] + 50)
cv2.rectangle(image, top_left, bottom_right, (0,0,255), 5)
cv2.imshow('Where is Waldo?', image)
```

Template matching is only useful when we have exact same image. scaling, distortion, rotation and changing brightness and hue will leave this technique useless so it is not ideal for most cases

Image Features - interesting area in an image that can describe an image also called key point features. eg - High change in intensity and corner in images

---

1 - Corner Detection

There are two method for corner detection

Harris coner detection

Harris Corner Detection is an algorithm developed in 1998 for corner detection (http://www.bmva.org/bmvc/1988/avc-88-023.pdf) and works fairly well.

cv2.cornerHarris(input image, block size, ksize, k)

- Input image - should be grayscale and float32 type.
- blockSize - the size of neighborhood considered for corner detection
- ksize - aperture parameter of Sobel derivative used.
- k - harris detector free parameter in the equation
- Output – array of corner locations (x,y)

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# The cornerHarris function requires the array datatype to be float32
gray = np.float32(gray)
harris_corners = cv2.cornerHarris(gray, 3, 3, 0.05)
#We use dilation of the corner points to enlarge them\
kernel = np.ones((7,7),np.uint8)
harris_corners = cv2.dilate(harris_corners, kernel, iterations = 2)
# Threshold for an optimal value, it may vary depending on the image.
image[harris_corners > 0.025 * harris_corners.max() ] = [255, 127, 127]
cv2.imshow('Harris Corners', image)
```

Good Features to Track

This is a improvement on Harris algo

cv2.goodFeaturesToTrack(input image, maxCorners, qualityLevel, minDistance)

- Input Image - 8-bit or floating-point 32-bit, single-channel image.
- maxCorners – Maximum number of corners to return. If there are more corners than are found, the strongest of them is returned.
- qualityLevel – Parameter characterizing the minimal accepted quality of image corners. The parameter value is multiplied by the best corner quality measure (smallest eigenvalue). The corners with the quality measure less than the product are rejected. For example, if the best corner has the quality measure = 1500, and the qualityLevel=0.01 , then all the corners with the quality - - measure less than 15 are rejected.
- minDistance – Minimum possible Euclidean distance between the returned corners.

```
# We specific the top 50 corners
corners = cv2.goodFeaturesToTrack(gray, 100, 0.01, 150)
```

```
corners = cv2.goodFeaturesToTrack(gray, 100, 0.01, 150)
for corner in corners:
    x, y = corner[0]
    x = int(x)
    y = int(y)
    cv2.rectangle(img,(x-10,y-10),(x+10,y+10),(0,255,0), 2)

cv2.imshow("Corners Found", img)
```

Coner detection has many problems

Corner matching in images is tolerant of:

- Rotations

- Translations (i.e. shifts in image)

- Slight photometric changes e.g. brightness

Scaling Issues

or affine intensity

However, it is intolerant of:

- Large changes in intensity or photometric changes

- Scaling (i.e. enlarging or shrinking)

---

2 - Features Detection (Sift, Surf, Fast, Brief ORB)

SIFT and SURF is now no longer free to use for

SIFT in a nutshell

We firstly detect interesting key points in an image using the Difference of Gaussian method These are areas of the image where variation exceeds a certain threshold and are better than edge descriptors.

- We then create vector descriptor for these interesting areas. Scale invariance is achieved via the following process:

- Interest points are scanned at several different scales

- The scale at which we meet a specific stability criteria, is then selected and is encoded by the vector descriptor. Therefore, regardless of the initial size, the more stable scale is found which allows us to be scale invariant.

- Rotation invariance is achieved by obtaining the Orientation Assignment of the key point using image gradient magnitudes. Once we know the 2D direction, we can normalize this direction.

- The full paper on SIFT can be read here: • http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

- An excellent tutorial on SIFT also available here

http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html

Speeded Up Robust Features (SURF)

SIFT is quite effective buy computationally expensive

SURF was developed to improve the speed of a scale invariant feature detector

Instead of using the Difference of Gaussian approach, SURF uses Hessian matrix approximation to detect interesting points and use the sum of Haar wavelet responses for orientation assignment.

Alternatives to SIFT and SURF

Features from Accelerated Segment Test (FAST)

• Key point detection only (no descriptor, we can use SIFT or SURF to computer that) • Used in real time applications

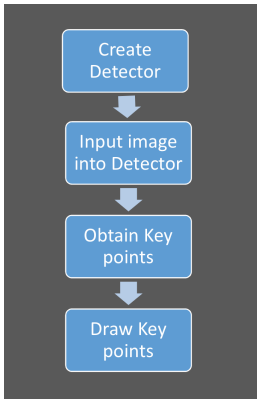
Binary Robust Independent Elementary Features (BRIEF)

• Computers descriptors quickly (instead of using SIFT or SURF) • Fast

Oriented FAST and Rotated BRIEF (ORB) – Developed out of OpenCV Labs (no patented so free to use!)

• Combines both Fast and Brief

• http://www.willowgarage.com/sites/default/files/orb_final.pdf



** pip install opencv-contrib-python **

SIFT

http://www.inf.fu-berlin.de/lehre/SS09/CV/uebungen/uebung09/SIFT.pdf

```
image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
#Create SIFT Feature Detector object
sift = cv2.xfeatures2d.SIFT_create()
#Detect key points
keypoints = sift.detect(gray, None)
print("Number of keypoints Detected: ", len(keypoints))
# Draw rich key points on input image
image = cv2.drawKeypoints(image, keypoints, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow('Feature Method - SIFT', image)
```

SURF

http://www.vision.ee.ethz.ch/~surf/eccv06.pdf

```
image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
#Create SURF Feature Detector object
surf = cv2.xfeatures2d.SURF_create()
# Only features, whose hessian is larger than hessianThreshold are retained by the detector
surf.setHessianThreshold(500)
keypoints, descriptors = surf.detectAndCompute(gray, None)
print("Number of keypoints Detected: ", len(keypoints))
# Draw rich key points on input image
image = cv2.drawKeypoints(image, keypoints, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow('Feature Method - SURF', image)
```

FAST

https://www.edwardrosten.com/work/rosten_2006_machine.pdf http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/AV1011

/AV1FeaturefromAcceleratedSegmentTest.pdf

```
image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Create FAST Detector object
fast = cv2.FastFeatureDetector_create()
# Obtain Key points, by default non max suppression is On
# to turn off set fast.setBool('nonmaxSuppression', False)
keypoints = fast.detect(gray, None)
print("Number of keypoints Detected: ", len(keypoints))
# Draw rich keypoints on input image
image = cv2.drawKeypoints(image, keypoints, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow('Feature Method - FAST', image)
```

BREIF

http://cvlabwww.epfl.ch/~lepetit/papers/calonder_pami11.pdf

```
image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Create FAST detector object
star = cv2.xfeatures2d.StarDetector_create()
# Create BRIEF extractor object
brief = cv2.xfeatures2d.BriefDescriptorExtractor_create()
# Determine key points
keypoints = star.detect(gray, None)
# Obtain descriptors and new final keypoints using BRIEF
keypoints, descriptors = brief.compute(gray, keypoints)
print("Number of keypoints Detected: ", len(keypoints))
# Draw rich keypoints on input image
image = cv2.drawKeypoints(image, keypoints,None,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow('Feature Method - BRIEF', image)
```

Oriented FAST and Rotated BRIEF (ORB)

http://www.willowgarage.com/sites/default/files/orb_final.pdf

```
image = cv2.imread('images/input.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Create ORB object, we can specify the number of key points we desire
orb = cv2.ORB_create()
# Determine key points
keypoints, descriptors = orb.detectAndCompute(gray, None)
# Obtain the descriptors
#keypoints, descriptors = orb.compute(gray, keypoints)
print("Number of keypoints Detected: ", len(keypoints))
# Draw rich keypoints on input image
image = cv2.drawKeypoints(image, keypoints, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imshow('Feature Method - ORB', image)
```

---

3 - Object Detection using ORB

Flannbased matching is used as it is quite fast but has low accuracy which is fine.

All the conept used in this code is already covered

```
import cv2
import numpy as np
def ORB_detector(new_image, image_template):
    # Function that compares input image to template
    # It then returns the number of ORB matches between them

    image1 = cv2.cvtColor(new_image, cv2.COLOR_BGR2GRAY)
    # Create ORB detector with 1000 keypoints with a scaling pyramid factor of 1.2
    orb = cv2.ORB_create(1000, 1.2)
    # Detect keypoints of original image
    kp1, des1 = orb.detectAndCompute(image1, None)
    # Detect keypoints of rotated image
    kp2, des2 = orb.detectAndCompute(image_template, None)
    # Create matcher
    # Note we're no longer using Flannbased matching
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    # Do matching
    matches = bf.match(des1,des2)
    # Sort the matches based on distance.  Least distance
    # is better
    matches = sorted(matches, key=lambda val: val.distance)
    return len(matches)
cap = cv2.VideoCapture(0)
# Load our image template, this is our reference image
image_template = cv2.imread('images/box_in_scene.png', 0)
# image_template = cv2.imread('images/kitkat.jpg', 0)
while True:
    # Get webcam images
    ret, frame = cap.read()

    # Get height and width of webcam frame
    height, width = frame.shape[:2]
    # Define ROI Box Dimensions (Note some of these things should be outside the loop)
    top_left_x = int(width / 3)
    top_left_y = int((height / 2) + (height / 4))
    bottom_right_x = int((width / 3) * 2)
    bottom_right_y = int((height / 2) - (height / 4))

    # Draw rectangular window for our region of interest
    cv2.rectangle(frame, (top_left_x,top_left_y), (bottom_right_x,bottom_right_y), (127,50,127),
3)

    # Crop window of observation we defined above
    cropped = frame[bottom_right_y:top_left_y , top_left_x:bottom_right_x]
    # Flip frame orientation horizontally
    frame = cv2.flip(frame,1)

    # Get number of ORB matches
    matches = ORB_detector(cropped, image_template)

    # Display status string showing the current no. of matches
    output_string = "Matches = " + str(matches)
    cv2.putText(frame, output_string, (50,450), cv2.FONT_HERSHEY_COMPLEX, 2, (250,0,150), 2)
```

```
    cv2.putText(frame, output_string, (50,450), cv2.FONT_HERSHEY_COMPLEX, 2, (250,0,150), 2)

    # Our threshold to indicate object deteciton
    # For new images or lightening conditions you may need to experiment a bit
    # Note: The ORB detector to get the top 1000 matches, 350 is essentially a min 35% match
    threshold = 350

    # If matches exceed our threshold then object has been detected
    if matches > threshold:
        cv2.rectangle(frame, (top_left_x,top_left_y), (bottom_right_x,bottom_right_y),
(0,255,0), 3)
        cv2.putText(frame,'Object Found',(50,50), cv2.FONT_HERSHEY_COMPLEX, 2 ,(0,255,0), 2)

    cv2.imshow('Object Detector using ORB', frame)

    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break
cap.release()
cv2.destroyAllWindows()
```

---

4 - Histogram of Oriented Gradients

- HOGs are a feature descriptor that has been widely and successfully used for object detection.

- In the HOG feature descriptor, the distribution (histograms) of directions of gradients (oriented gradients) are used as features. Gradients (x and y derivatives) of an image are useful because the magnitude of gradients is large around edges and corners (regions of abrupt intensity changes) and we know that edges and corners pack in a lot more information about object shape than flat regions.

- It represents objects as a single feature vector as opposed to a set of feature vectors where each represents a segment of the image.

- It's computed by sliding window detector over an image, where a HOG descriptor is a computed for each position. Like SIFT the scale of the image is adjusted (pyramiding).

- HOGs are often used with SVM (support vector machine) classifiers. Each HOG descriptor that is computed is fed to a SVM classifier to determine if the object was found or not).

- read Paper by Dalal & Triggs on using HOGs for Human Detection: https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
# Load image then grayscale
image = cv2.imread('images/elephant.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Show original Image
cv2.imshow('Input Image', image)
cv2.waitKey(0)
# h x w in pixels
cell_size = (8, 8)
# h x w in cells
block_size = (2, 2)
# number of orientation bins
nbins = 9
# Using OpenCV's HOG Descriptor
# winSize is the size of the image cropped to a multiple of the cell size
```

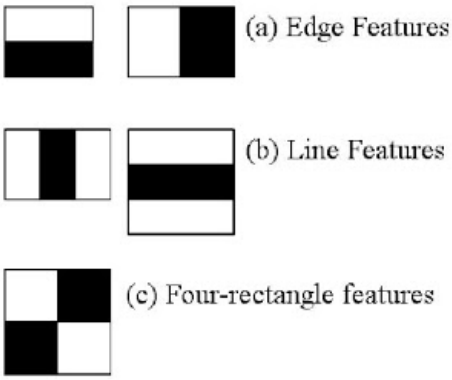
```
hog = cv2.HOGDescriptor(_winSize=(gray.shape[1] // cell_size[1] * cell_size[1],
                                  gray.shape[0] // cell_size[0] * cell_size[0]),
                        _blockSize=(block_size[1] * cell_size[1],
                                    block_size[0] * cell_size[0]),
                        _blockStride=(cell_size[1], cell_size[0]),
                        _cellSize=(cell_size[1], cell_size[0]),
                        _nbins=nbins)
# Create numpy array shape which we use to create hog_feats
n_cells = (gray.shape[0] // cell_size[0], gray.shape[1] // cell_size[1])
# We index blocks by rows first.
# hog_feats now contains the gradient amplitudes for each direction,
# for each cell of its group for each group. Indexing is by rows then columns.
hog_feats = hog.compute(gray).reshape(n_cells[1] - block_size[1] + 1,
                        n_cells[0] - block_size[0] + 1,
                        block_size[0], block_size[1], nbins).transpose((1, 0, 2, 3, 4))
# Create our gradients array with nbin dimensions to store gradient orientations
gradients = np.zeros((n_cells[0], n_cells[1], nbins))
# Create array of dimensions
cell_count = np.full((n_cells[0], n_cells[1], 1), 0, dtype=int)
# Block Normalization
for off_y in range(block_size[0]):
    for off_x in range(block_size[1]):
        gradients[off_y:n_cells[0] - block_size[0] + off_y + 1,
                  off_x:n_cells[1] - block_size[1] + off_x + 1] += \
            hog_feats[:, :, off_y, off_x, :]
        cell_count[off_y:n_cells[0] - block_size[0] + off_y + 1,
                   off_x:n_cells[1] - block_size[1] + off_x + 1] += 1
# Average gradients
gradients /= cell_count
# Plot HOGs using Matplotlib
# angle is 360 / nbins * direction
color_bins = 5
plt.pcolor(gradients[:, :, color_bins])
plt.gca().invert_yaxis()
plt.gca().set_aspect('equal', adjustable='box')
plt.colorbar()
plt.show()
cv2.destroyAllWindows()
```

Haar Cascade Classifiers

Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.

Here we will work with face detection. Initially, the algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. Then we need to extract features from it. For this, Haar features shown in the below

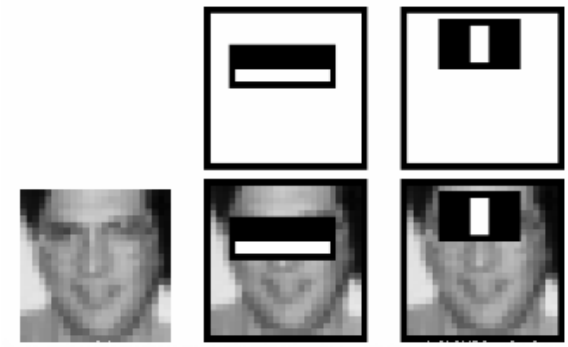
image are used. They are just like our convolutional kernel. Each feature is a single value obtained by subtracting sum of pixels under the white rectangle from sum of pixels under the black rectangle.



image

Now, all possible sizes and locations of each kernel are used to calculate lots of features. (Just imagine how much computation it needs? Even a 24x24 window results over 160000 features). For each feature calculation, we need to find the sum of the pixels under white and black rectangles. To solve this, they introduced the integral image. However large your image, it reduces the calculations for a given pixel to an operation involving just four pixels. Nice, isn't it? It makes things super-fast.

But among all these features we calculated, most of them are irrelevant. For example, consider the image below. The top row shows two good features. The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks. The second feature selected relies on the property that the eyes are darker than the bridge of the nose. But the same windows applied to cheeks or any other place is irrelevant. So how do we select the best features out of 160000+ features? It is achieved by Adaboost.



image

For this, we apply each and every feature on all the training images. For each feature, it finds the best threshold which will classify the faces to positive and negative. Obviously, there will be errors or misclassifications. We select the features with minimum error rate, which means they are the features that most accurately classify the face and non-face images. (The process is not as simple as this. Each image is given an equal weight in the beginning. After each classification, weights of misclassified images are increased. Then the same process is done. New error rates are calculated. Also new weights. The process is continued until the required accuracy or error rate is achieved or the required number of features are found).

The final classifier is a weighted sum of these weak classifiers. It is called weak because it alone can't classify the image, but together with others forms a strong classifier. The paper says even 200 features provide detection with 95% accuracy. Their final setup had around 6000 features. (Imagine a reduction from 160000+ features to 6000 features. That is a big gain).

So now you take an image. Take each 24x24 window. Apply 6000 features to it. Check if it is face or not. Wow.. Isn't it a little inefficient and time consuming? Yes, it is. The authors have a good solution for that.

In an image, most of the image is non-face region. So it is a better idea to have a simple method to check if a window is not a face region. If it is not, discard it in a single shot, and don't process it again. Instead, focus on regions where there can be a face. This way, we spend more time checking possible face regions.

For this they introduced the concept of Cascade of Classifiers. Instead of applying all 6000 features on a window, the features are grouped into different stages of classifiers and applied one-by-one. (Normally the first few stages will contain very many fewer features). If a window fails the first stage, discard it. We don't consider the remaining features on it. If it passes, apply the second stage of features and continue the process. The window which passes all stages is a face region. How is that plan!

The authors' detector had 6000+ features with 38 stages with 1, 10, 25, 25 and 50 features in the first five stages. (The two features in the above image are actually obtained as the best two features from Adaboost). According to the authors, on average 10 features out of 6000+ are evaluated per sub-window.

So this is a simple intuitive explanation of how Viola-Jones face detection works. Read the paper for more details or check out the references in the Additional Resources section.

haarcascade_car.xml	119 kB
haarcascade_eye.xml	341 kB
haarcascade_frontalface_default.xml	930 kB
haarcascade_fullbody.xml	477 kB

1- Face & Eye Detection using HAAR Cascade Classifiers

```
import numpy as np
import cv2
# We point OpenCV's CascadeClassifier function to where our
# classifier (XML file format) is stored
face_classifier = cv2.CascadeClassifier('Haarcascades/haarcascade_frontalface_default.xml')
# Load our image then convert it to grayscale
image = cv2.imread('images/Trump.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Our classifier returns the ROI of the detected face as a tuple
# It stores the top left coordinate and the bottom right coordiantes
faces = face_classifier.detectMultiScale(gray, 1.3, 5)
# When no faces detected, face_classifier returns and empty tuple
if faces is ():
    print("No faces found")
# We iterate through our faces array and draw a rectangle
# over each face in faces
for (x,y,w,h) in faces:
    cv2.rectangle(image, (x,y), (x+w,y+h), (127,0,255), 2)
    cv2.imshow('Face Detection', image)
    cv2.waitKey(0)

cv2.destroyAllWindows()
```

2- Let's combine face and eye detection

2- Let's combine face and eye detection

```
import numpy as np
import cv2
face_classifier = cv2.CascadeClassifier('Haarcascades/haarcascade_frontalface_default.xml')
eye_classifier = cv2.CascadeClassifier('Haarcascades/haarcascade_eye.xml')
img = cv2.imread('images/Trump.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_classifier.detectMultiScale(gray, 1.3, 5)
# When no faces detected, face_classifier returns and empty tuple
if faces is ():
    print("No Face Found")
for (x,y,w,h) in faces:
    cv2.rectangle(img,(x,y),(x+w,y+h),(127,0,255),2)
    cv2.imshow('img',img)
    cv2.waitKey(0)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    eyes = eye_classifier.detectMultiScale(roi_gray)
    for (ex,ey,ew,eh) in eyes:
        cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(255,255,0),2)
        cv2.imshow('img',img)
        cv2.waitKey(0)

cv2.destroyAllWindows()
```

3- Let's make a live face & eye detection, keeping the face inview at all times

```
import cv2
import numpy as np
face_classifier = cv2.CascadeClassifier('Haarcascades/haarcascade_frontalface_default.xml')
eye_classifier = cv2.CascadeClassifier('Haarcascades/haarcascade_eye.xml')
def face_detector(img, size=0.5):
    # Convert image to grayscale
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    faces = face_classifier.detectMultiScale(gray, 1.3, 5)
    if faces is ():
        return img

    for (x,y,w,h) in faces:
        x = x - 50
        w = w + 50
        y = y - 50
        h = h + 50
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = img[y:y+h, x:x+w]
        eyes = eye_classifier.detectMultiScale(roi_gray)

        for (ex,ey,ew,eh) in eyes:
            cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,0,255),2)

    roi_color = cv2.flip(roi_color,1)
    return roi_color
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    cv2.imshow('Our Face Extractor', face_detector(frame))
```

```
    cv2.imshow('Our Face Extractor', face_detector(frame))
    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break

cap.release()
cv2.destroyAllWindows()
```

4- Pedistrian Detection

```
import cv2
import numpy as np
# Create our body classifier
body_classifier = cv2.CascadeClassifier('Haarcascades/haarcascade_fullbody.xml')
# Initiate video capture for video file
cap = cv2.VideoCapture('images/walking.avi')
# Loop once video is successfully loaded
while cap.isOpened():

    # Read first frame
    ret, frame = cap.read()
    frame = cv2.resize(frame, None,fx=0.5, fy=0.5, interpolation = cv2.INTER_LINEAR)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Pass frame to our body classifier
    bodies = body_classifier.detectMultiScale(gray, 1.2, 3)

    # Extract bounding boxes for any bodies identified
    for (x,y,w,h) in bodies:
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 255), 2)
        cv2.imshow('Pedestrians', frame)
    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break
cap.release()
cv2.destroyAllWindows()
```

Car Detection

```
import cv2
import time
import numpy as np
# Create our body classifier
car_classifier = cv2.CascadeClassifier('Haarcascades/haarcascade_car.xml')
# Initiate video capture for video file
cap = cv2.VideoCapture('images/cars.avi')
# Loop once video is successfully loaded
while cap.isOpened():

    time.sleep(.05)
    # Read first frame
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Pass frame to our car classifier
    cars = car_classifier.detectMultiScale(gray, 1.4, 2)

    # Extract bounding boxes for any bodies identified
    for (x,y,w,h) in cars:
```

```
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 255), 2)
        cv2.imshow('Cars', frame)
    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break
cap.release()
cv2.destroyAllWindows()
```

---

1 - Creating Haar Cascade

Training a haar cascade is lot diff than training a neural net as i previously assumed. U just have to collect data and put it in two folders one with positve and other with negative images then train the haar cascade by running terminal command which train a haar cascade for u, u don't have create any form of python code for this.

Getting Negative images:

This is for the negatives images, blackground images in other words.

Here are a few links for these images, you can download each one these images, resize it and convert it to grey scale.

http://image-net.org/api/text/imagenet.synset.geturls?wnid=n07942152

http://image-net.org/api/text/imagenet.synset.geturls?wnid=n00523513

```
import urllib.request
import cv2
import numpy as np
import os
def store_raw_images():
    neg_images_link = 'http://image-net.org/api/text/imagenet.synset.geturls?wnid=n00523513'
    neg_image_urls = urllib.request.urlopen(neg_images_link).read().decode()
    pic_num = 1

    if not os.path.exists('neg'):
        os.makedirs('neg')

    for i in neg_image_urls.split('\n'):
        try:
            print(i)
            urllib.request.urlretrieve(i, "neg/"+str(pic_num)+".jpg")
            img = cv2.imread("neg/"+str(pic_num)+".jpg",cv2.IMREAD_GRAYSCALE)
            # should be larger than samples / pos pic (so we can place our image on it)
            resized_image = cv2.resize(img, (100, 100))
            cv2.imwrite("neg/"+str(pic_num)+".jpg",resized_image)
            pic_num += 1

        except Exception as e:
            print(str(e))
store_raw_images()
```

Running this script will create a new directory '***neg***' with a few hundreds of images.

Note : when you are downloading from several ***URLS***, make sure to change the ***pic_num*** in the script is ***'number of images in neg folder + 1'***

You will override images if you don't update ***'pic_num'.***

Cleaning images and creating description files :
Sometimes when the images are removed from the website, it shows a ***'this image has been removed'*** picture, instead of a broken link or a not found error and this image gets downloaded.

Here is the script to remove these images from our '***neg***' folder.

```
def find_uglies():
    match = False
    for file_type in ['neg']:
        for img in os.listdir(file_type):
            for ugly in os.listdir('uglies'):
                try:
                    current_image_path = str(file_type)+'/'+str(img)
                    ugly = cv2.imread('uglies/'+str(ugly))
                    question = cv2.imread(current_image_path)
                    if ugly.shape == question.shape and not(np.bitwise_xor(ugly,question).any()):
                        print('That is one ugly pic! Deleting!')
                        print(current_image_path)
                        os.remove(current_image_path)
                except Exception as e:
                    print(str(e))
find_uglies()
```

Before running this script, create a new folder ***'uglies'***
Which holds the ***'image has been removed'*** image (referred as ugly image) or any other image which you want to remove.
Also note that this step is optional, not removing these unwanted images may not cause any problems.

How are we checking if the images in '***uglies***' and images in '***neg***' as same ?

```
if ugly.shape == question.shape and not(np.bitwise_xor(ugly,question).any()):
```

Optionally you can use ***compare ssim*** :

```
from skimage.measure import structural_similarity as ssim
s = ssim(imageA, imageB)
print (s)
```

```
print (e)
```

If this condition is true, the delete that image!

Now to get the ***positives*** / the actual image we want to detect.
Also, make sure to have at least 1000+ negative images for better accuracy.

Run the ***store_raw_images()*** function with different urls (once again don't forget to update ***pic_num***)

create the descriptor file for these negative images:

```
def create_pos_n_neg():
    for file_type in ['neg']:

        for img in os.listdir(file_type):
            if file_type == 'pos':
                line = file_type+'/'+img+' 1 0 0 50 50\n'
                with open('info.dat','a') as f:
                    f.write(line)
            elif file_type == 'neg':
                line = file_type+'/'+img+'\n'
                with open('bg.txt','a') as f:
                    f.write(line)
```

After running this script a new file '***bg.txt***' will be generated which holds the file name of all the negative images we downloaded and converted!

Here is how your ***bg.txt*** would look like :

```
neg/1.jpg
neg/10.jpg
neg/100.jpg
neg/1000.jpg
neg/1001.jpg
neg/1002.jpg
neg/1003.jpg
neg/1007.jpg
neg/1008.jpg
neg/1009.jpg
neg/101.jpg
neg/1010.jpg
neg/1011.jpg
neg/1012.jpg
neg/1013.jpg
neg/1015.jpg
neg/1016.jpg
neg/1017.jpg
neg/1018.jpg
neg/1019.jpg
neg/102.jpg
```

Creating samples (positives):

Create two more folders '***data***' and '***info***',
Where ***data*** holds the ***cascade*** file and ***info*** holds samples of positive images.
creating sample from a ***single image*** using ***create_samples (opencv)***
Note : When we use ***create_samples*** method from ***opencv***, Our object detector can only detect that particular object we train for (single image).
Say that single image is your watch, the it can only detect your watch and nothing else (not even other watches).

To create positive samples :

```
opencv_createsamples -img custom_image.jpg -bg bg.txt -info info/info.lst -pngoutput info -
maxxangle 0.5 -maxyangle 0.5 -maxzangle 0.5 -num 1950
```

Where '***info***' holds positive sample images and ***info.lst***, ***info.lst*** is a file similar to ***bg.txt*** for positive samples.
Note : if you get an error '***invalid background description file***'
It's either because '***bg.txt***' file is corrupted or improper installation of ***opencv.***

1) CTRL + A ; CTRL + C; on your background description file to copy all it's content.

2) Create a new background description file (e.g.: bg2.txt), and CTRL + V to paste all the content you copied. Save it.

3) Run your command again, using, of course, the new bg description file (-bg bg2.txt).

The ***info*** folder now contains images which '***single image***' ***superimposed*** on negative images (images in ***neg*** folder).

Now to create the vector file :

```
opencv_createsamples -info info/info.lst -num 1950 -w 20 -h 20 -vec positives.vec
```

This creates a new file '***positives.vec***'
Note that images are now taken to be '***20×20***', only because it would take much longer to train.

I would definitely recommend to try bigger images!

Training the cascade :
Brace yourself, this is going a take a lot of time.

This is the current file structure :

```
opencv_workspace
--neg
----negimages.jpg
--opencv
--info
--data
--positives.vec --bg.txt
--single_image.jpg
```

Train command :

Create a new folder ***'data'***

```
opencv_traincascade -data data -vec positives.vec -bg bg.txt -numPos 1800 -numNeg 900 -numStages 10 -w 20 -h 20
```

Note : as the number of stages increases, better is the accuracy and not to forget

Longer the training process.

As said earlier, it is a convention to select positives – twice the number of negatives.

This may take up to 30 minutes to 3 hours of time, depending on your machine

If you want to ***run this process in the background***, Use this command :

```
nohup opencv_traincascade -data data -vec positives.vec -bg bg.txt -numPos 1800 -numNeg 900 -numStages 10 -w 20 -h 20 &
```

The best part of ***opencv_traincascade*** is that, you can cancel your training anytime (***stagex.xml*** has to be generated) and continue with the training later by running the exact same command.

Once the training is complete, data folder would contain ***x number of 'stagex.xml'*** files

Errors you might come accross :

Required leaf false alarm rate achieved. Branch training terminated.

which may be because, there is a feature that perfectly separates positives from negatives, so you get HR=1 and FA=0.

Insufficient samples.

Clearly because the number of positives are not sufficient for training more stages (solution would be to reduce number of stages or increase number of samples)

After completing all the stages, ***'cascade.xml'*** will be generated.

You can also combine several cascade files for detection.

---

2 - Face Analysis Using Haar Cascade

Finding Facial landmark using Dlib - download Dlib - https://sourceforge.net/projects/dclib/

Download the pre-trained model here - http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2

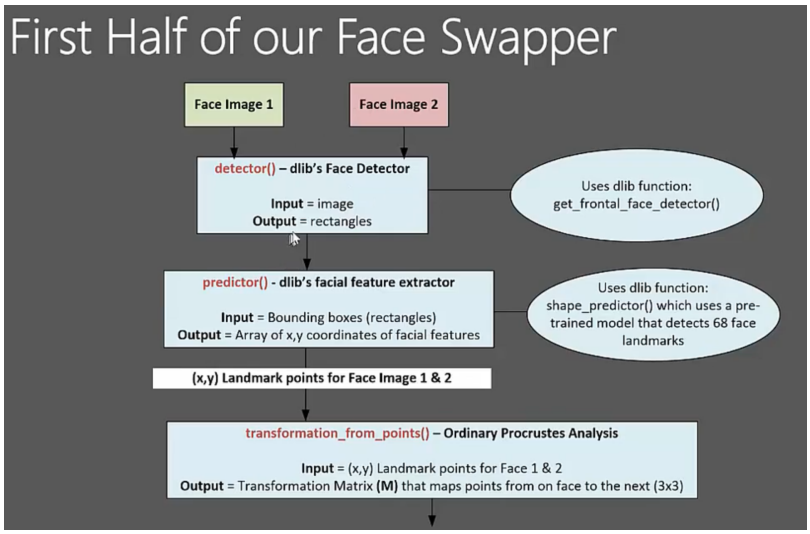
Face Morphing - face swapping

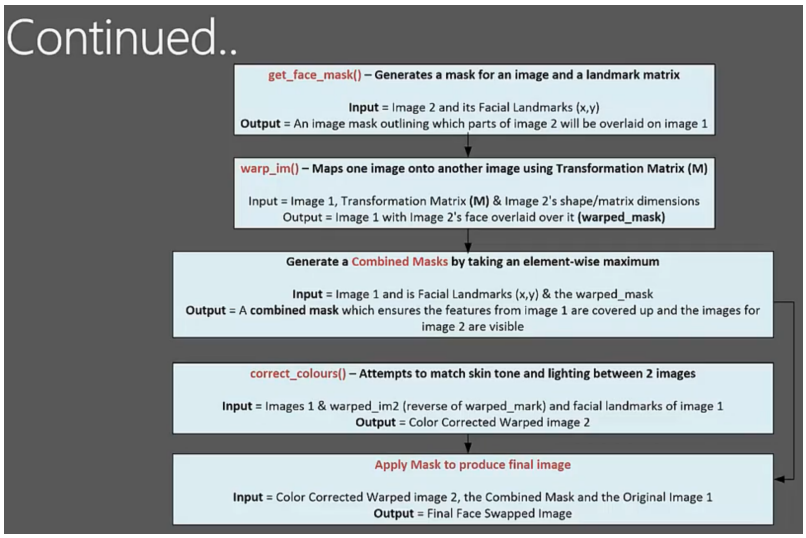
1- Detecting Facial landmark

2 - Wraping the image to fit new face

3 - Color Matching

4 - Creating seamless border





Facial Lamdmark Detection Code

```
import cv2
import dlib
import numpy
PREDICTOR_PATH = "shape_predictor_68_face_landmarks.dat"
predictor = dlib.shape_predictor(PREDICTOR_PATH)
detector = dlib.get_frontal_face_detector()
```

```
class TooManyFaces(Exception):
    pass
class NoFaces(Exception):
    pass
def get_landmarks(im):
    rects = detector(im, 1)
    if len(rects) > 1:
        raise TooManyFaces
    if len(rects) == 0:
        raise NoFaces
    return numpy.matrix([[p.x, p.y] for p in predictor(im, rects[0]).parts()])
def annotate_landmarks(im, landmarks):
    im = im.copy()
    for idx, point in enumerate(landmarks):
        pos = (point[0, 0], point[0, 1])
        cv2.putText(im, str(idx), pos,
                    fontFace=cv2.FONT_HERSHEY_SCRIPT_SIMPLEX,
                    fontScale=0.4,

                    color=(0, 0, 255))
        cv2.circle(im, pos, 3, color=(0, 255, 255))
    return im
image = cv2.imread('./images/Obama.jpg')
cv2.imshow("test", image)
cv2.waitKey(0)
landmarks = get_landmarks(image)
image_with_landmarks = annotate_landmarks(image, landmarks)
cv2.imshow('Result', image_with_landmarks)
cv2.imwrite('image_with_landmarks.jpg',image_with_landmarks)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Face Swap Code

```
import cv2
import dlib
import numpy
from time import sleep
import sys
PREDICTOR_PATH = "shape_predictor_68_face_landmarks.dat"
SCALE_FACTOR = 1
FEATHER_AMOUNT = 11
FACE_POINTS = list(range(17, 68))
MOUTH_POINTS = list(range(48, 61))
RIGHT_BROW_POINTS = list(range(17, 22))
LEFT_BROW_POINTS = list(range(22, 27))
RIGHT_EYE_POINTS = list(range(36, 42))
LEFT_EYE_POINTS = list(range(42, 48))
NOSE_POINTS = list(range(27, 35))
JAW_POINTS = list(range(0, 17))
# Points used to line up the images.
ALIGN_POINTS = (LEFT_BROW_POINTS + RIGHT_EYE_POINTS + LEFT_EYE_POINTS +
                               RIGHT_BROW_POINTS + NOSE_POINTS + MOUTH_POINTS)
# Points from the second image to overlay on the first. The convex hull of each
# element will be overlaid.
OVERLAY_POINTS = [
    LEFT_EYE_POINTS + RIGHT_EYE_POINTS + LEFT_BROW_POINTS + RIGHT_BROW_POINTS
```

```
    LEFT_EYE_POINTS + RIGHT_EYE_POINTS + LEFT_BROW_POINTS + RIGHT_BROW_POINTS,
    NOSE_POINTS + MOUTH_POINTS,
]
# Amount of blur to use during colour correction, as a fraction of the
# pupillary distance.
COLOUR_CORRECT_BLUR_FRAC = 0.6
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor(PREDICTOR_PATH)
class TooManyFaces(Exception):
    pass
class NoFaces(Exception):
    pass
def get_landmarks(im):
    # Returns facial landmarks as (x,y) coordinates
    rects = detector(im, 1)

    if len(rects) > 1:
        raise TooManyFaces
    if len(rects) == 0:
        raise NoFaces
    return numpy.matrix([[p.x, p.y] for p in predictor(im, rects[0]).parts()])
def annotate_landmarks(im, landmarks):
    #Overlays the landmark points on the image itself

    im = im.copy()
    for idx, point in enumerate(landmarks):
        pos = (point[0, 0], point[0, 1])
        cv2.putText(im, str(idx), pos,
                    fontFace=cv2.FONT_HERSHEY_SCRIPT_SIMPLEX,
                    fontScale=0.4,
                    color=(0, 0, 255))
        cv2.circle(im, pos, 3, color=(0, 255, 255))
    return im
def draw_convex_hull(im, points, color):
    points = cv2.convexHull(points)
    cv2.fillConvexPoly(im, points, color=color)
def get_face_mask(im, landmarks):
    im = numpy.zeros(im.shape[:2], dtype=numpy.float64)
    for group in OVERLAY_POINTS:
        draw_convex_hull(im,
                         landmarks[group],
                         color=1)
    im = numpy.array([im, im, im]).transpose((1, 2, 0))
    im = (cv2.GaussianBlur(im, (FEATHER_AMOUNT, FEATHER_AMOUNT), 0) > 0) * 1.0
    im = cv2.GaussianBlur(im, (FEATHER_AMOUNT, FEATHER_AMOUNT), 0)
    return im

def transformation_from_points(points1, points2):
    """
    Return an affine transformation [s * R | T] such that:
        sum ||s*R*p1,i + T - p2,i||^2
    is minimized.
    """
    # Solve the procrustes problem by subtracting centroids, scaling by the
    # standard deviation, and then using the SVD to calculate the rotation. See
    # the following for more details:
    #   https://en.wikipedia.org/wiki/Orthogonal_Procrustes_problem
    points1 = points1.astype(numpy.float64)
```

```
    points2 = points2.astype(numpy.float64)
    c1 = numpy.mean(points1, axis=0)
    c2 = numpy.mean(points2, axis=0)
    points1 -= c1
    points2 -= c2
    s1 = numpy.std(points1)
    s2 = numpy.std(points2)
    points1 /= s1
    points2 /= s2
    U, S, Vt = numpy.linalg.svd(points1.T * points2)
    # The R we seek is in fact the transpose of the one given by U * Vt. This
    # is because the above formulation assumes the matrix goes on the right
    # (with row vectors) where as our solution requires the matrix to be on the
    # left (with column vectors).
    R = (U * Vt).T
    return numpy.vstack([numpy.hstack(((s2 / s1) * R,
                                       c2.T - (s2 / s1) * R * c1.T)),
                         numpy.matrix([0., 0., 1.])])
def read_im_and_landmarks(image):
    im = image
    im = cv2.resize(im,None,fx=1, fy=1, interpolation = cv2.INTER_LINEAR)
    im = cv2.resize(im, (im.shape[1] * SCALE_FACTOR,
                         im.shape[0] * SCALE_FACTOR))
    s = get_landmarks(im)
    return im, s
def warp_im(im, M, dshape):
    output_im = numpy.zeros(dshape, dtype=im.dtype)
    cv2.warpAffine(im,
                   M[:2],
                   (dshape[1], dshape[0]),
                   dst=output_im,
                   borderMode=cv2.BORDER_TRANSPARENT,
                   flags=cv2.WARP_INVERSE_MAP)
    return output_im
def correct_colours(im1, im2, landmarks1):
    blur_amount = COLOUR_CORRECT_BLUR_FRAC * numpy.linalg.norm(
                              numpy.mean(landmarks1[LEFT_EYE_POINTS], axis=0) -
                              numpy.mean(landmarks1[RIGHT_EYE_POINTS], axis=0))
    blur_amount = int(blur_amount)
    if blur_amount % 2 == 0:
        blur_amount += 1
    im1_blur = cv2.GaussianBlur(im1, (blur_amount, blur_amount), 0)
    im2_blur = cv2.GaussianBlur(im2, (blur_amount, blur_amount), 0)
    # Avoid divide-by-zero errors.
    im2_blur += (128 * (im2_blur <= 1.0)).astype(im2_blur.dtype)
    return (im2.astype(numpy.float64) * im1_blur.astype(numpy.float64) /
                                                im2_blur.astype(numpy.float64))
def swappy(image1, image2):

    im1, landmarks1 = read_im_and_landmarks(image1)
    im2, landmarks2 = read_im_and_landmarks(image2)
    M = transformation_from_points(landmarks1[ALIGN_POINTS],
                                   landmarks2[ALIGN_POINTS])

    mask = get_face_mask(im2, landmarks2)
    warped_mask = warp_im(mask, M, im1.shape)
    combined_mask = numpy.max([get_face_mask(im1, landmarks1), warped_mask],
```

```
                               axis=0)
    warped_im2 = warp_im(im2, M, im1.shape)
    warped_corrected_im2 = correct_colours(im1, warped_im2, landmarks1)
    output_im = im1 * (1.0 - combined_mask) + warped_corrected_im2 * combined_mask
    cv2.imwrite('output.jpg', output_im)
    image = cv2.imread('output.jpg')
    return image

## Enter the paths to your input images here
image1 = cv2.imread('images/Hillary.jpg')
image2 = cv2.imread('images/Trump.jpg')
swapped = swappy(image1, image2)
cv2.imshow('Face Swap 1', swapped)
swapped = swappy(image2, image1)
cv2.imshow('Face Swap 2', swapped)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Live Face Swapping

```
import cv2
import dlib
import numpy
from time import sleep
import sys
## Our pretrained model that predicts the rectangles that correspond to the facial features of a
face
PREDICTOR_PATH = "shape_predictor_68_face_landmarks.dat"
SCALE_FACTOR = 1
FEATHER_AMOUNT = 11
FACE_POINTS = list(range(17, 68))
MOUTH_POINTS = list(range(48, 61))
RIGHT_BROW_POINTS = list(range(17, 22))
LEFT_BROW_POINTS = list(range(22, 27))
RIGHT_EYE_POINTS = list(range(36, 42))
LEFT_EYE_POINTS = list(range(42, 48))
NOSE_POINTS = list(range(27, 35))
JAW_POINTS = list(range(0, 17))
# Points used to line up the images.
ALIGN_POINTS = (LEFT_BROW_POINTS + RIGHT_EYE_POINTS + LEFT_EYE_POINTS +
                               RIGHT_BROW_POINTS + NOSE_POINTS + MOUTH_POINTS)
# Points from the second image to overlay on the first. The convex hull of each
# element will be overlaid.
OVERLAY_POINTS = [
    LEFT_EYE_POINTS + RIGHT_EYE_POINTS + LEFT_BROW_POINTS + RIGHT_BROW_POINTS,
    NOSE_POINTS + MOUTH_POINTS,
]
# Amount of blur to use during colour correction, as a fraction of the
# pupillary distance.
COLOUR_CORRECT_BLUR_FRAC = 0.6
cascade_path='Haarcascades/haarcascade_frontalface_default.xml'
cascade = cv2.CascadeClassifier(cascade_path)
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor(PREDICTOR_PATH)
def get_landmarks(im, dlibOn):
```

```
    if (dlibOn == True):
        rects = detector(im, 1)
        if len(rects) > 1:
            return "error"
        if len(rects) == 0:
            return "error"
        return numpy.matrix([[p.x, p.y] for p in predictor(im, rects[0]).parts()])

    else:
        rects = cascade.detectMultiScale(im, 1.3,5)
        if len(rects) > 1:
            return "error"
        if len(rects) == 0:
            return "error"
        x,y,w,h =rects[0]
        rect=dlib.rectangle(x,y,x+w,y+h)
        return numpy.matrix([[p.x, p.y] for p in predictor(im, rect).parts()])

def annotate_landmarks(im, landmarks):
    im = im.copy()
    for idx, point in enumerate(landmarks):
        pos = (point[0, 0], point[0, 1])
        cv2.putText(im, str(idx), pos,
                    fontFace=cv2.FONT_HERSHEY_SCRIPT_SIMPLEX,
                    fontScale=0.4,
                    color=(0, 0, 255))
        cv2.circle(im, pos, 3, color=(0, 255, 255))
    return im
def draw_convex_hull(im, points, color):
    points = cv2.convexHull(points)
    cv2.fillConvexPoly(im, points, color=color)

def get_face_mask(im, landmarks):
    im = numpy.zeros(im.shape[:2], dtype=numpy.float64)
    for group in OVERLAY_POINTS:
        draw_convex_hull(im,
                         landmarks[group],
                         color=1)
    im = numpy.array([im, im, im]).transpose((1, 2, 0))
    im = (cv2.GaussianBlur(im, (FEATHER_AMOUNT, FEATHER_AMOUNT), 0) > 0) * 1.0
    im = cv2.GaussianBlur(im, (FEATHER_AMOUNT, FEATHER_AMOUNT), 0)
    return im


def transformation_from_points(points1, points2):
    """
    Return an affine transformation [s * R | T] such that:
        sum ||s*R*p1,i + T - p2,i||^2
    is minimized.
    """
    # Solve the procrustes problem by subtracting centroids, scaling by the
    # standard deviation, and then using the SVD to calculate the rotation. See
    # the following for more details:
    #   https://en.wikipedia.org/wiki/Orthogonal_Procrustes_problem
    points1 = points1.astype(numpy.float64)
    points2 = points2.astype(numpy.float64)
```

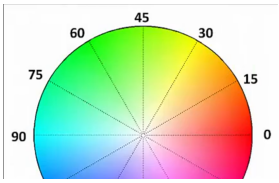
```
    c1 = numpy.mean(points1, axis=0)
    c2 = numpy.mean(points2, axis=0)
    points1 -= c1
    points2 -= c2
    s1 = numpy.std(points1)
    s2 = numpy.std(points2)
    points1 /= s1
    points2 /= s2
    U, S, Vt = numpy.linalg.svd(points1.T * points2)
    # The R we seek is in fact the transpose of the one given by U * Vt. This
    # is because the above formulation assumes the matrix goes on the right
    # (with row vectors) where as our solution requires the matrix to be on the
    # left (with column vectors).
    R = (U * Vt).T
    return numpy.vstack([numpy.hstack(((s2 / s1) * R,
                                       c2.T - (s2 / s1) * R * c1.T)),
                         numpy.matrix([0., 0., 1.])])
def read_im_and_landmarks(fname):
    im = cv2.imread(fname, cv2.IMREAD_COLOR)
    im = cv2.resize(im,None,fx=0.35, fy=0.35, interpolation = cv2.INTER_LINEAR)
    im = cv2.resize(im, (im.shape[1] * SCALE_FACTOR,
                         im.shape[0] * SCALE_FACTOR))
    s = get_landmarks(im,dlibOn)
    return im, s
def warp_im(im, M, dshape):
    output_im = numpy.zeros(dshape, dtype=im.dtype)
    cv2.warpAffine(im,
                   M[:2],
                   (dshape[1], dshape[0]),
                   dst=output_im,
                   borderMode=cv2.BORDER_TRANSPARENT,
                   flags=cv2.WARP_INVERSE_MAP)
    return output_im
def correct_colours(im1, im2, landmarks1):
    blur_amount = COLOUR_CORRECT_BLUR_FRAC * numpy.linalg.norm(
                              numpy.mean(landmarks1[LEFT_EYE_POINTS], axis=0) -
                              numpy.mean(landmarks1[RIGHT_EYE_POINTS], axis=0))
    blur_amount = int(blur_amount)
    if blur_amount % 2 == 0:
        blur_amount += 1
    im1_blur = cv2.GaussianBlur(im1, (blur_amount, blur_amount), 0)
    im2_blur = cv2.GaussianBlur(im2, (blur_amount, blur_amount), 0)
    # Avoid divide-by-zero errors.
    im2_blur += (128 * (im2_blur <= 1.0)).astype(im2_blur.dtype)
    return (im2.astype(numpy.float64) * im1_blur.astype(numpy.float64) /
                                                im2_blur.astype(numpy.float64))
def face_swap(img,name):
    s = get_landmarks(img,True)

    if (s == "error"):
        print("No or too many faces")
        return img

    im1, landmarks1 = img, s
    im2, landmarks2 = read_im_and_landmarks(name)
    M = transformation_from_points(landmarks1[ALIGN_POINTS],
                                   landmarks2[ALIGN_POINTS])
    mask = get_face_mask(im2, landmarks2)
```

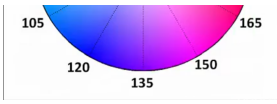
```
    mask = get_face_mask(im2, landmarks2)
    warped_mask = warp_im(mask, M, im1.shape)
    combined_mask = numpy.max([get_face_mask(im1, landmarks1), warped_mask],
                              axis=0)
    warped_im2 = warp_im(im2, M, im1.shape)

    warped_corrected_im2 = correct_colours(im1, warped_im2, landmarks1)
    output_im = im1 * (1.0 - combined_mask) + warped_corrected_im2 * combined_mask

    #output_im is no longer in the expected OpenCV format so we use openCV
    #to write the image to diks and then reload it
    cv2.imwrite('output.jpg', output_im)
    image = cv2.imread('output.jpg')

    frame = cv2.resize(image,None,fx=1.5, fy=1.5, interpolation = cv2.INTER_LINEAR)

    return image

cap = cv2.VideoCapture(0)
# Name is the image we want to swap onto ours
# dlibOn controls if use dlib's facial landmark detector (better)
# or use HAAR Cascade Classifiers (faster)
filter_image =  "images/Trump.jpg" ### Put your image here!
dlibOn = False
while True:
    ret, frame = cap.read()

    #Reduce image size by 75% to reduce processing time and improve framerates
    frame = cv2.resize(frame, None, fx=0.75, fy=0.75, interpolation = cv2.INTER_LINEAR)

    # flip image so that it's more mirror like
    frame = cv2.flip(frame, 1)

    cv2.imshow('Our Amazing Face Swapper', face_swap(frame, filter_image))

    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break
cap.release()
cv2.destroyAllWindows()
```

Motion Detection

1 - Color Filtering

We use Hue to filter out colours. First we convert the image to HSV then filter out colours.





Hue colour range, goes from 0 to 180

- Red - 165 to 15
- Green - 45 to 75
- Blue - 90 to 120

Saturation and Value/Brightness range

- Typically we set the filter range 50 to 255 for both saturation and value as this capture true colour
- 0 to 60 in Saturation is close to white
- 0 to 60 in value is close to black

```
import cv2
import numpy as np
# Initialize webcam
cap = cv2.VideoCapture(0)
# define range of PURPLE color in HSV
lower_purple = np.array([125,0,0])
upper_purple = np.array([175,255,255])
# loop until break statement is exectured
while True:
    # Read webcam image
    ret, frame = cap.read()

    # Convert image from RBG/BGR to HSV so we easily filter
    hsv_img = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    # Use inRange to capture only the values between lower & upper_blue
    mask = cv2.inRange(hsv_img, lower_purple, upper_purple)
    # Perform Bitwise AND on mask and our original frame
    res = cv2.bitwise_and(frame, frame, mask=mask)
    cv2.imshow('Original', frame)
    cv2.imshow('mask', mask)
    cv2.imshow('Filtered Color Only', res)
    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break

cap.release()
cv2.destroyAllWindows()
```

---

2 - Background Subtraction

This is a very useful technique which allow us to remove separate background from foreground in a video stream. These algorithm able to learn about the frame in view and able to accurately "learn" and identify foreground. Which result in a binary segmentation of

able to learn about the frame in view and able to accurately "learn" and identify foreground. Which result in a binary segmentation of the image which highlight the moving objects.

Background subtraction algorithm

- Background Subtractor MOG
- Background Subtractor MOG2
- Geometric Multigrid (GMG)

1. Gaussian Mixture-based Background/Foreground Segmentation Algorithm

```
# OpenCV 2.4.13 only
import numpy as np
import cv2



cap = cv2.VideoCapture('./images/walking.avi')



# Initlaize background subtractor
foreground_background = cv2.bgsegm.createBackgroundSubtractorMOG()
while True:
    ret, frame = cap.read()
    # Apply background subtractor to get our foreground mask
    foreground_mask = foreground_background.apply(frame)
    cv2.imshow('Output', foreground_mask)




    if cv2.waitKey(1) == 13:
        break
cap.release()
cv2.destroyAllWindows()
```

Let's try the Improved adaptive Gausian mixture model for background subtraction

```
# OpenCV 2.4.13
import numpy as np
import cv2


cap = cv2.VideoCapture('./images/walking.avi')


# Initlaize background subtractor
foreground_background = cv2.bgsegm.createBackgroundSubtractorGSOC()


while True:
    ret, frame = cap.read()
    # Apply background subtractor to get our foreground mask
```

```
    # Apply background subtractor to get our foreground mask
    foreground_mask = foreground_background.apply(frame)
    cv2.imshow('Output', foreground_mask)


    if cv2.waitKey(1) == 13:
        break


cap.release()
cv2.destroyAllWindows()
```

What about foreground substraction?Background Substraction KKN

---

3 - Object Tracking Using - MeanShift

The idea behind **meanshift** is that in meanshift algorithm every instance of the video is checked in the form of pixel distribution in that frame. We define an initial window, generally a square or a circle for which the positions are specified by ourself which identifies the area of maximum pixel distribution and tries to keep track of that area in the video so that when the video is running our tracking window also moves towards the region of maximum pixel distribution. The direction of movement depends upon the difference between the center of our tracking window and the centroid of all the k-pixels inside that window.

```
import numpy as np
import cv2
# Initialize webcam
cap = cv2.VideoCapture(0)
# take first frame of the video
ret, frame = cap.read()
print(type(frame))
# setup default location of window
r, h, c, w = 240, 100, 400, 160
track_window = (c, r, w, h)
# Crop region of interest for tracking
roi = frame[r:r+h, c:c+w]
# Convert cropped window to HSV color space
hsv_roi =  cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
# Create a mask between the HSV bounds
lower_purple = np.array([125,0,0])
upper_purple = np.array([175,255,255])
mask = cv2.inRange(hsv_roi, lower_purple, upper_purple)
# Obtain the color histogram of the ROI
roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0,180])
# Normalize values to lie between the range 0, 255
cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)
# Setup the termination criteria
# We stop calculating the centroid shift after ten iterations
# or if the centroid has moved at least 1 pixel
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )
```

```
while True:
    # Read webcam frame
    ret, frame = cap.read()
    if ret == True:
        # Convert to HSV
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)


        # Calculate the histogram back projection
        # Each pixel's value is it's probability
        dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)
        # apply meanshift to get the new location
        ret, track_window = cv2.meanShift(dst, track_window, term_crit)
        # Draw it on image
        x, y, w, h = track_window
        img2 = cv2.rectangle(frame, (x,y), (x+w, y+h), 255, 2)
        cv2.imshow('Meansift Tracking', img2)

        if cv2.waitKey(1) == 13: #13 is the Enter Key
            break
    else:
        break
cv2.destroyAllWindows()
cap.release()
```

---

4 - Object Tracking Using - CamShift

Camshift is very similar to meanshift, however u have noted that meanshift uses fixed window size for object tracking which can be problematic. Camshift uses adaptive window size that changes both orientation and size.
Camshift uses meanshift untill it converges and calculate the size and orientation of the window.

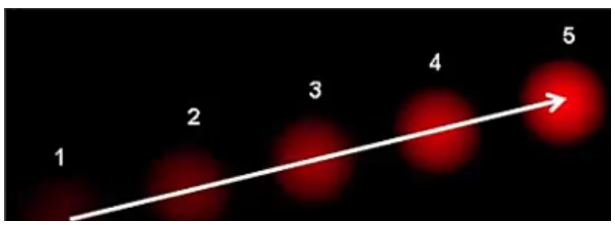
```
import numpy as np
import cv2
# Initialize webcam
cap = cv2.VideoCapture(0)
# take first frame of the video
ret, frame = cap.read()
# setup default location of window
r, h, c, w = 240, 100, 400, 160
track_window = (c, r, w, h)
# Crop region of interest for tracking
roi = frame[r:r+h, c:c+w]
# Convert cropped window to HSV color space
hsv_roi =  cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
# Create a mask between the HSV bounds
lower_purple = np.array([130,60,60])
upper_purple = np.array([175,255,255])
mask = cv2.inRange(hsv_roi, lower_purple, upper_purple)
```

```
# Obtain the color histogram of the ROI
roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0,180])
# Normalize values to lie between the range 0, 255
cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)
# Setup the termination criteria
# We stop calculating the centroid shift after ten iterations
# or if the centroid has moved at least 1 pixel
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )
while True:

    # Read webcam frame
    ret, frame = cap.read()
    if ret == True:
        # Convert to HSV
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

        # Calculate the histogram back projection
        # Each pixel's value is it's probability
        dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)


        # apply Camshift to get the new location
        ret, track_window = cv2.CamShift(dst, track_window, term_crit)


        # Draw it on image
        # We use polylines to represent Adaptive box
        pts = cv2.boxPoints(ret)
        pts = np.int0(pts)
        img2 = cv2.polylines(frame,[pts],True, 255,2)

        cv2.imshow('Camshift Tracking', img2)

        if cv2.waitKey(1) == 13: #13 is the Enter Key
            break
    else:
        break
cv2.destroyAllWindows()
cap.release()
```

---

5 - Optical Flow

Optical flow seeks to get the pattern of object movement in an image between two frames. It show the distribution of the apparent velocities of object in an image.



There are two implementation of optical flow -

- Lucas-Kanade Differential Method - track only some point, good for tracking corner feature
- Dense Optical Flow - Slower but track all feature, colors are used to track also

Lucas-Kanade Optical Flow in OpenCV

```
import numpy as np
import cv2
# Load video stream
cap = cv2.VideoCapture('images/walking.avi')
# Set parameters for ShiTomasi corner detection
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )
# Set parameters for lucas kanade optical flow
lucas_kanade_params = dict( winSize  = (15,15),
                  maxLevel = 2,
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))
# Create some random colors
# Used to create our trails for object movement in the image
color = np.random.randint(0,255,(100,3))
# Take first frame and find corners in it
ret, prev_frame = cap.read()
prev_gray = cv2.cvtColor(prev_frame, cv2.COLOR_BGR2GRAY)
# Find inital corner locations
prev_corners = cv2.goodFeaturesToTrack(prev_gray, mask = None, **feature_params)
# Create a mask image for drawing purposes
mask = np.zeros_like(prev_frame)
while(1):
    ret, frame = cap.read()
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # calculate optical flow
    new_corners, status, errors = cv2.calcOpticalFlowPyrLK(prev_gray,
                                                           frame_gray,
                                                           prev_corners,
                                                           None,
                                                           **lucas_kanade_params)
    # Select and store good points
    good_new = new_corners[status==1]
    good_old = prev_corners[status==1]
    # Draw the tracks
    for i,(new,old) in enumerate(zip(good_new, good_old)):
        a, b = new.ravel()
        c, d = old.ravel()
        mask = cv2.line(mask, (a,b),(c,d), color[i].tolist(), 2)
        frame = cv2.circle(frame, (a,b), 5, color[i].tolist(),-1)

    img = cv2.add(frame,mask)
    # Show Optical Flow
    cv2.imshow('Optical Flow - Lucas-Kanade',img)
    if cv2.waitKey(1) == 13: #13 is the Enter Key
```

```
        break
    # Now update the previous frame and previous points
    prev_gray = frame_gray.copy()
    prev_corners = good_new.reshape(-1,1,2)
cv2.destroyAllWindows()
cap.release()
```

Dense Optical Flow

```
import cv2
import numpy as np
# Load video stream
cap = cv2.VideoCapture("images/walking.avi")
# Get first frame
ret, first_frame = cap.read()
previous_gray = cv2.cvtColor(first_frame, cv2.COLOR_BGR2GRAY)
hsv = np.zeros_like(first_frame)
hsv[...,1] = 255
while True:

    # Read of video file
    ret, frame2 = cap.read()
    next = cv2.cvtColor(frame2,cv2.COLOR_BGR2GRAY)
    # Computes the dense optical flow using the Gunnar Farneback's algorithm
    flow = cv2.calcOpticalFlowFarneback(previous_gray, next,
                                        None, 0.5, 3, 15, 3, 5, 1.2, 0)
    # use flow to calculate the magnitude (speed) and angle of motion
    # use these values to calculate the color to reflect speed and angle
    magnitude, angle = cv2.cartToPolar(flow[...,0], flow[...,1])
    hsv[...,0] = angle * (180 / (np.pi/2))
    hsv[...,2] = cv2.normalize(magnitude, None, 0, 255, cv2.NORM_MINMAX)
    final = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)
    # Show our demo of Dense Optical Flow
    cv2.imshow('Dense Optical Flow', final)
    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break

    # Store current image as previous image
    previous_gray = next
cap.release()
cv2.destroyAllWindows()
```

---

5 - Object Tracking Final

```
# USAGE
# python opencv_object_tracking.py
# python opencv_object_tracking.py --video dashcam_boston.mp4 --tracker csrt


# i
```

```
# import the necessary packages
from imutils.video import VideoStream
from imutils.video import FPS
import argparse
import imutils
import time
import cv2


# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video", type=str,
    help="path to input video file")
ap.add_argument("-t", "--tracker", type=str, default="kcf",
    help="OpenCV object tracker type")
args = vars(ap.parse_args())


# extract the OpenCV version info
(major, minor) = cv2.__version__.split(".")[:2]


# if we are using OpenCV 3.2 OR BEFORE, we can use a special factory
# function to create our object tracker
if int(major) == 3 and int(minor) < 3:
    tracker = cv2.Tracker_create(args["tracker"].upper())


# otherwise, for OpenCV 3.3 OR NEWER, we need to explicity call the
# approrpiate object tracker constructor:
else:
    # initialize a dictionary that maps strings to their corresponding
    # OpenCV object tracker implementations
    OPENCV_OBJECT_TRACKERS = {
        "csrt": cv2.TrackerCSRT_create,
        "kcf": cv2.TrackerKCF_create,
        "boosting": cv2.TrackerBoosting_create,
        "mil": cv2.TrackerMIL_create,
        "tld": cv2.TrackerTLD_create,
        "medianflow": cv2.TrackerMedianFlow_create,
        "mosse": cv2.TrackerMOSSE_create
    }


    # grab the appropriate object tracker using our dictionary of
    # OpenCV object tracker objects
    tracker = OPENCV_OBJECT_TRACKERS[args["tracker"]]()


# initialize the bounding box coordinates of the object we are going
# to track
initBB = None


# if a video path was not supplied, grab the reference to the web cam
if not args.get("video", False):
    print("[INFO] starting video stream...")
    vs = VideoStream(src=0).start()
```

```
    vs = VideoStream(src=0).start()
    time.sleep(1.0)


# otherwise, grab a reference to the video file
else:
    vs = cv2.VideoCapture(args["video"])


# initialize the FPS throughput estimator
fps = None


# loop over frames from the video stream
while True:
    # grab the current frame, then handle if we are using a
    # VideoStream or VideoCapture object
    frame = vs.read()
    frame = frame[1] if args.get("video", False) else frame


    # check to see if we have reached the end of the stream
```