ALWAYS REFER THE OFFICIAL DOCUMENTATION OF OPENCV:
https://docs.opencv.org/2.4/

Recommended Online Course:
https://www.udacity.com/course/introduction-to-computer-vision--ud810

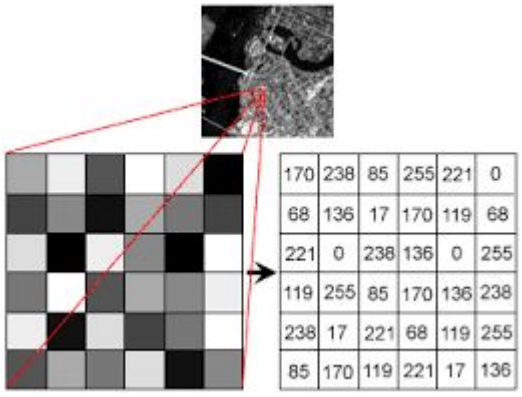
FOLLOWING CONTENT COPIED FROM THE BLOG:
https://www.analyticsvidhya.com/blog/2019/03/opencv-functions-computer-vision-python/

OTHER SIMILAR BLOGS: http://www.robindavid.fr/opencv-tutorial/

Table of Contents

Reading, Writing and Displaying Images

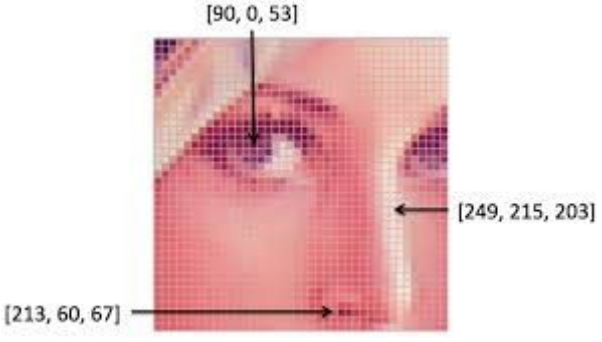
Machines see and process everything using numbers, including images and text. How do you convert images to numbers – I can hear you wondering. Two words – pixel values:



Every number represents the pixel intensity at that particular location. In the above image, I have shown the pixel values for a grayscale image where every pixel contains only one value i.e. the intensity of the black color at that location.

Note that color images will have multiple values for a single pixel. These values represent the intensity of respective channels – Red, Green and Blue channels for RGB images, for instance.

Reading and writing images is essential to any computer vision project. And the OpenCV library makes this function a whole lot easier.



Now, let's see how to import an image into our machine using OpenCV. Download the image from here.

```
#import the libraries

import numpy as np

import matplotlib.pyplot as plt

import cv2

%matplotlib inline



#reading the image



image = cv2.imread('index.png')

image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

#plotting the image

plt.imshow(image)



#saving image

cv2.imwrite('test_write.jpg',image)
```

By default, the *imread* function reads images in the BGR (Blue-Green-Red) format. We can read images in different formats using extra flags in the *imread* function:

- **cv2.IMREAD_COLOR:** Default flag for loading a color image
- **cv2.IMREAD_GRAYSCALE:** Loads images in grayscale format
- **cv2.IMREAD_UNCHANGED:** Loads images in their given format, including the alpha channel. Alpha channel stores the transparency information – the higher the value of alpha channel, the more opaque is the pixel

Changing Color Spaces

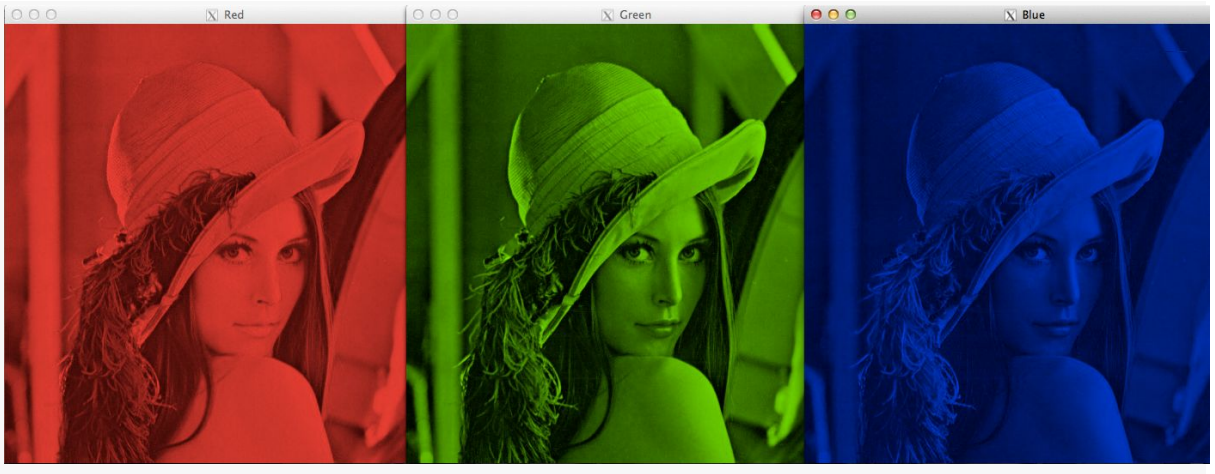
A color space is a protocol for representing colors in a way that makes them easily reproducible. We know that grayscale images have single pixel values and color images contain 3 values for each pixel – the intensities of the Red, Green and Blue channels.

Most computer vision use cases process images in RGB format. However, applications like video compression and device independent storage – these are heavily dependent on other color spaces, like the Hue-Saturation-Value or HSV color space.

As you understand a RGB image consists of the color intensity of different color channels, i.e. the intensity and color information are mixed in RGB color space but in HSV color space the color and intensity information are separated from each other. This makes HSV color space more robust to lighting changes.

OpenCV reads a given image in the BGR format by default. So, you'll need to change the color space of your image from BGR to RGB when reading images using OpenCV. Let's see how to do that:

```
#import the required libraries

import numpy as np

import matplotlib.pyplot as plt

import cv2

%matplotlib inline

image = cv2.imread('index.jpg')

#converting image to Gray scale

gray_image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)

#plotting the grayscale image

plt.imshow(gray_image)

#converting image to HSV format

hsv_image = cv2.cvtColor(image,cv2.COLOR_BGR2HSV)

#plotting the HSV image

plt.imshow(hsv_image)
```



Resizing Images

Machine learning models work with a fixed sized input. The same idea applies to computer vision models as well. The images we use for training our model must be of the same size.

Now this might become problematic if we are creating our own dataset by scraping images from various sources. That's where the function of resizing images comes to the fore.

Images can be easily scaled up and down using OpenCV. This operation is useful for training deep learning models when we need to convert images to the model's input shape. Different interpolation and downsampling methods are supported by OpenCV, which can be used by the following parameters:

1. **INTER_NEAREST:** Nearest neighbor interpolation
2. **INTER_LINEAR:** Bilinear interpolation
3. **INTER_AREA:** Resampling using pixel area relation
4. **INTER_CUBIC:** Bicubic interpolation over 4×4 pixel neighborhood
5. **INTER_LANCZOS4:** Lanczos interpolation over 8×8 neighborhood

OpenCV's resize function uses bilinear interpolation by default.

```
import cv2

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

#reading the image

image = cv2.imread('index.jpg')

#converting image to size (100,100,3)

smaller_image = cv2.resize(image,(100,100),inerpolation='linear')
```

```
#plot the resized image

plt.imshow(smaller_image)
```

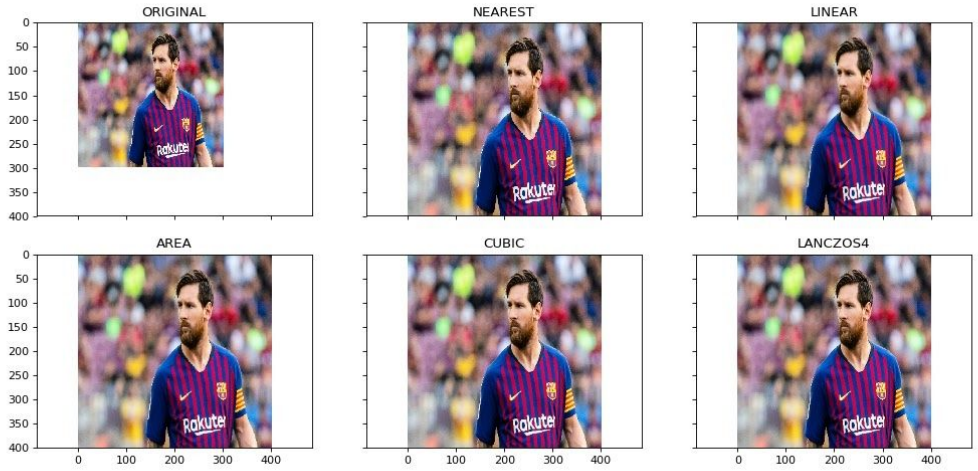


Image Rotation

“You need a large amount of data to train a deep learning model”. I’m sure you must have comes across this line of thought in form or another. It’s partially true – most deep learning algorithms are heavily dependent on the quality and quantity of the data.

But what if you do not have a large enough dataset? Not all of us can afford to manually collect and label images.

Suppose we are building an image classification model for identifying the animal present in an image. So, both the images shown below should be classified as ‘dog’:

But the model might find it difficult to classify the second image as a Dog if it was not trained on such images. So what should we do?

Let me introduce you to the technique of data augmentation. This method allows us to generate more samples for training our deep learning model. Data augmentation uses the available data samples to produce the new ones, by applying image operations like rotation, scaling, translation, etc. This makes our model robust to changes in input and leads to better generalization.

Rotation is one of the most used and easy to implement data augmentation techniques. As the name suggests, it involves rotating the image at an arbitrary angle and providing it the same label as the original image. Think of the times you have rotated images in your phone to achieve certain angles – that's basically what this function does.

```
#importing the required libraries

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline

image = cv2.imread('index.png')

rows,cols = image.shape[:2]

#(col/2,rows/2) is the center of rotation for the image

# M is the cordinates of the center

M = cv2.getRotationMatrix2D((cols/2,rows/2),90,1)

dst = cv2.warpAffine(image,M,(cols,rows))

plt.imshow(dst)
```

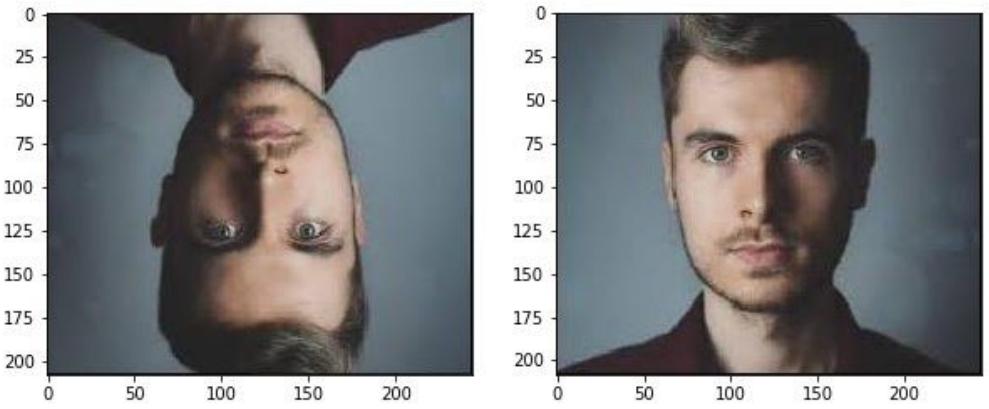


Image Translation

Image translation is a geometric transformation that maps the position of every object in the image to a new location in the final output image. After the translation operation, an object present at location (x,y) in the input image is shifted to a new position (X,Y):

$$X = x + dx$$

$$Y = y + dy$$

Here, dx and dy are the respective translations along different dimensions.

Image translation can be used to add shift invariance to the model, as by tranlation we can change the position of the object in the image give more variety to the model that leads to better generalizability which works in difficult conditions i.e. when the object is not perfectly aligned to the center of the image.

This augmentation technique can also help the model correctly classify images with partially visible objects. Take the below image for example. Even when the complete shoe is not present in the image, the model should be able to classify it as a Shoe.

This translation function is typically used in the image pre-processing stage. Check out the below code to see how it works in a practical scenario:

```
#importing the required libraries

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline

#reading the image

image = cv2.imread('index.png')

#shifting the image 100 pixels in both dimensions

M = np.float32([[1,0,-100],[0,1,-100]])

dst = cv2.warpAffine(image,M,(cols,rows))

plt.imshow(dst)
```

Simple Image Thresholding

Thresholding is an image **segmentation** method. It compares pixel values with a threshold value and updates it accordingly. OpenCV supports multiple variations of thresholding. A simple thresholding function can be defined like this:

if Image(x,y) > threshold , Image(x,y) = 1

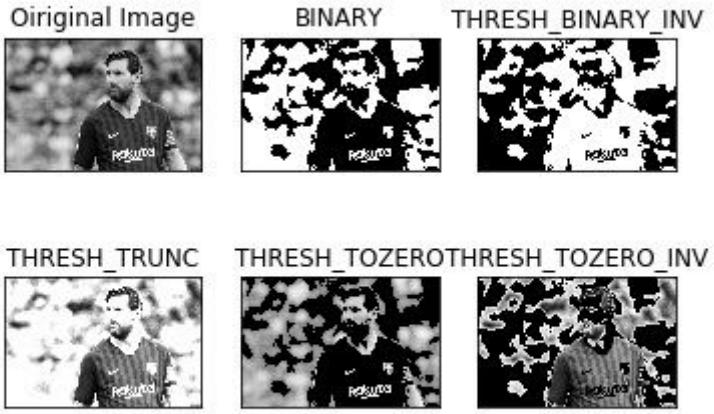
otherswise, Image(x,y) = 0

Thresholding can only be applied to grayscale images.

A simple application of image thresholding could be dividing the image into it's foreground and background.

```
#importing the required libraries

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline




#here 0 means that the image is loaded in gray scale format

gray_image = cv2.imread('index.png',0)




ret,thresh_binary = cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)

ret,thresh_binary_inv =
cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY_INV)
```

```
ret,thresh_trunc = cv2.threshold(gray_image,127,255,cv2.THRESH_TRUNC)

ret,thresh_tozero = cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO)

ret,thresh_tozero_inv =
cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO_INV)



#DISPLAYING THE DIFFERENT THRESHOLDING STYLES

names = ['Oiriginal
Image','BINARY','THRESH_BINARY_INV','THRESH_TRUNC','THRESH_TOZERO','THRESH_TO
ZERO_INV']

images =
gray_image,thresh_binary,thresh_binary_inv,thresh_trunc,thresh_tozero,thresh_
tozero_inv



for i in range(6):

   plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')

   plt.title(names[i])

   plt.xticks([]),plt.yticks([])



plt.show()
```



Adaptive Thresholding

In case of adaptive thresholding, different threshold values are used for different parts of the image. This function gives better results for images with varying lighting conditions – hence the term “adaptive”.

Otsu’s binarization method finds an optimal threshold value for the whole image. It works well for bimodal images (images with 2 peaks in their histogram).

```
#import the libraries

import numpy as np

import matplotlib.pyplot as plt

import cv2

%matplotlib inline




#ADAPTIVE THRESHOLDING

gray_image = cv2.imread('index.png',0)
```

```
ret,thresh_global = cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)

#here 11 is the pixel neighbourhood that is used to calculate the threshold
value

thresh_mean =
cv2.adaptiveThreshold(gray_image,255,cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BI
NARY,11,2)



thresh_gaussian =
cv2.adaptiveThreshold(gray_image,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRES
H_BINARY,11,2)



names = ['Original Image','Global Thresholding','Adaptive Mean
Threshold','Adaptive Gaussian Thresholding']

images = [gray_image,thresh_global,thresh_mean,thresh_gaussian]



for i in range(4):

   plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')

   plt.title(names[i])

   plt.xticks([]),plt.yticks([])
```

```
plt.show()
```

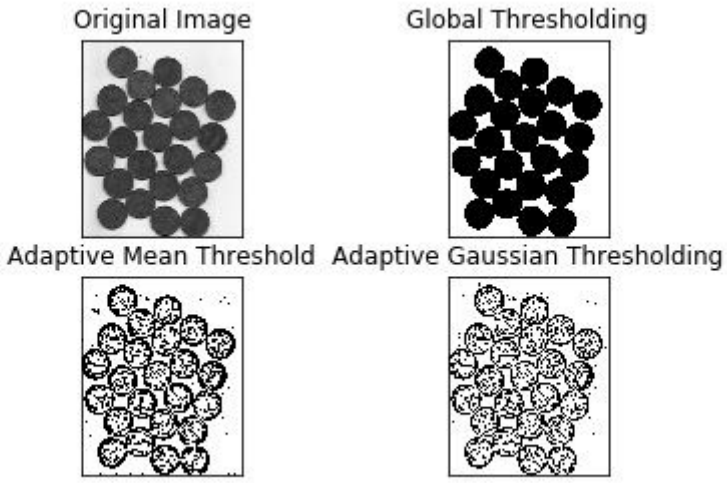


Image Segmentation (Watershed Algorithm)

Image segmentation is the task of classifying every pixel in the image to some class. For example, classifying every pixel as foreground or background. Image segmentation is important for extracting the relevant parts from an image.

The watershed algorithm is a classic image segmentation algorithm. It considers the pixel values in an image as topography. For finding the object boundaries, it takes initial markers as input. The algorithm then starts flooding the basin from the markers till the markers meet at the object boundaries.

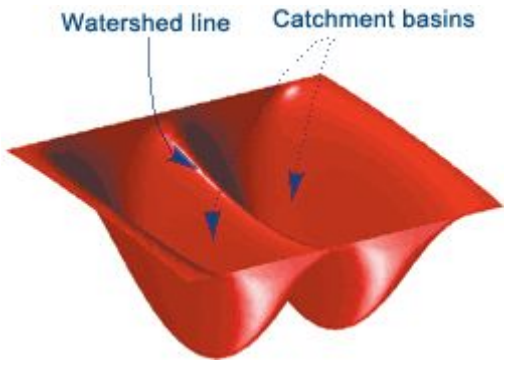
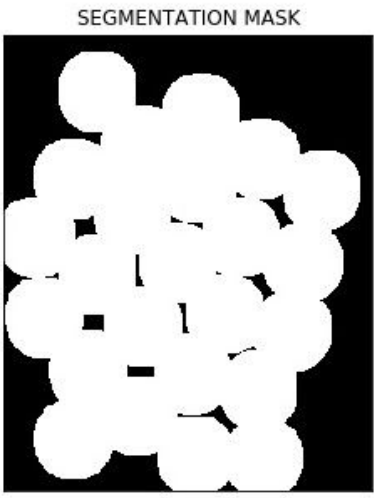


Image Source :- Mathworks

Let's say we have a topography with multiple basins. Now, if we fill different basins with water of different color, then the intersection of different colors will give us the object boundaries. This is the intuition behind the watershed algorithm.

```
#importing required libraries

import numpy as np

import cv2

import matplotlib.pyplot as plt




#reading the image

image = cv2.imread('coins.jpg')

#converting image to grayscale format

gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)

#apply thresholding

ret,thresh =
cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

#get a kernel

kernel = np.ones((3,3),np.uint8)

opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel,iterations = 2)

#extract the background from image
```

```
sure_bg = cv2.dilate(opening,kernel,iterations = 3)



dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)

ret,sure_fg = cv2.threshold(dist_transform,0.7*dist_transform.max(),255,0)



sure_fg = np.uint8(sure_fg)

unknown = cv2.subtract(sure_bg,sure_bg)



ret,markers = cv2.connectedComponents(sure_fg)



markers = markers+1



markers[unknown==255] = 0



markers = cv2.watershed(image,markers)

image[markers==-1] = [255,0,0]



plt.imshow(sure_fg)
```

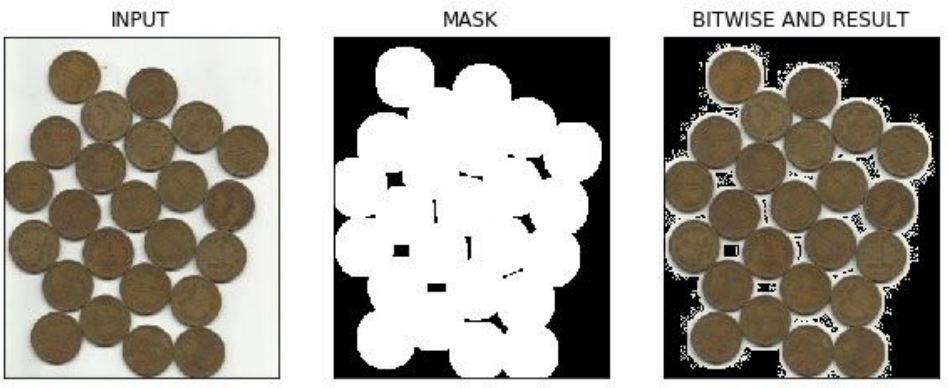


Bitwise Operations

Bitwise operations include AND, OR, NOT and XOR. You might remember them from your programming class! In computer vision, these operations are very useful when we have a mask image and want to apply that mask over another image to extract the region of interest.

```
#import required libraries

import numpy as np

import matplotlib.pyplot as plt

import cv2

%matplotlib inline

#read the image

image = cv2.imread('coins.jpg')
```

```
#apply thresholdin

ret,mask =
cv2.threshold(sure_fg,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

#apply AND operation on image and mask generated by thrresholding

final = cv2.bitwise_and(image,image,mask = mask)

#plot the result

plt.imshow(final)
```



In the above figure, we can see an input image and its segmentation mask calculated using the Watershed algorithm. Further, we have applied the bitwise 'AND' operation to remove the background from the image and extract relevant portions from the image. Pretty awesome stuff!

Edge Detection

Edges are the points in an image where the image brightness changes sharply or has discontinuities. Such discontinuities generally correspond to:

- Discontinuities in depth
- Discontinuities in surface orientation
- Changes in material properties
- Variations in scene illumination

Edges are very useful features of an image that can be used for different applications like classification of objects in the image and localization. Even deep learning models calculate edge features to extract information about the objects present in image.

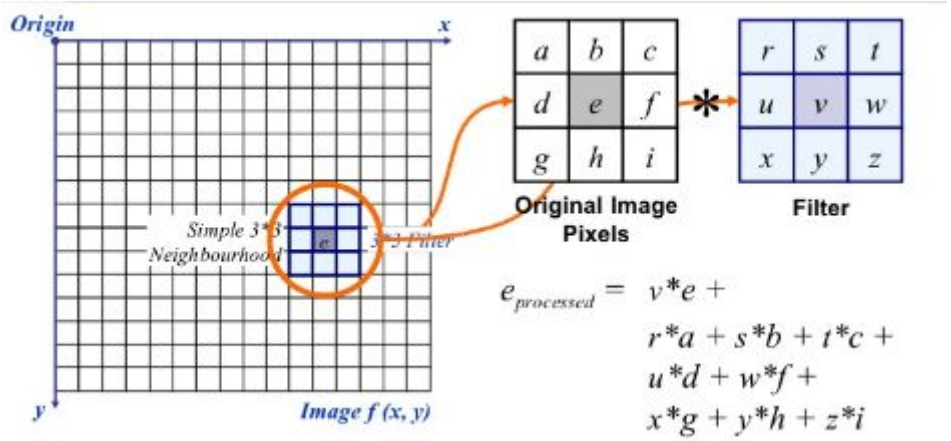
Edges are different from contours as they are not related to objects rather they signify the changes in pixel values of an image. Edge detection can be used for image segmentation and even for image sharpening.

```
#import the required libraries

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline

#read the image

image = cv2.imread('coins.jpg')

#calculate the edges using Canny edge algorithm

edges = cv2.Canny(image,100,200)

#plot the edges

plt.imshow(edges)
```

Image Filtering

In image filtering, a pixel value is updated using its neighbouring values. But how are these values updated in the first place?

Well, there are multiple ways of updating pixel values, such as selecting the maximum value from neighbours, using the average of neighbours, etc. Each method has it's own uses. For example, averaging the pixel values in a neighbourhood is used for image blurring.



Gaussian filtering is also used for image blurring that gives different weights to the neighbouring pixels based on their distance from the pixel under consideration.

For image filtering, we use kernels. Kernels are matrices of numbers of different shapes like 3 x 3, 5 x 5, etc. A kernel is used to calculate the dot product with a part of the image. When calculating the new value of a pixel, the kernel center is overlapped with the pixel. The neighbouring pixel values are multiplied with the corresponding values in the kernel. The calculated value is assigned to the pixel coinciding with the center of the kernel.

```
#importing the required libraries

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline

image = cv2.imread('index.png')

#using the averaging kernel for image smoothening

averaging_kernel = np.ones((3,3),np.float32)/9

filtered_image = cv2.filter2D(image,-1,kernel)

plt.imshow(dst)

#get a one dimensional Gaussian Kernel

gaussian_kernel_x = cv2.getGaussianKernel(5,1)

gaussian_kernel_y = cv2.getGaussianKernel(5,1)

#converting to two dimensional kernel using matrix multiplication

gaussian_kernel = gaussian_kernel_x * gaussian_kernel_y.T
```

```
#you can also use cv2.GaussianBLurring(image,(shape of kernel),standard
deviation) instead of cv2.filter2D

filtered_image = cv2.filter2D(image,-1,gaussian_kernel)

plt.imshow()
```

In the above output, the image on the right shows the result of applying Gaussian kernels on an input image. We can see that the edges of the original image are suppressed. The Gaussian kernel with different values of sigma is used extensively to calculate the Difference of Gaussian for our image. **This is an important step in the feature extraction process because it reduces the noise present in the image.**

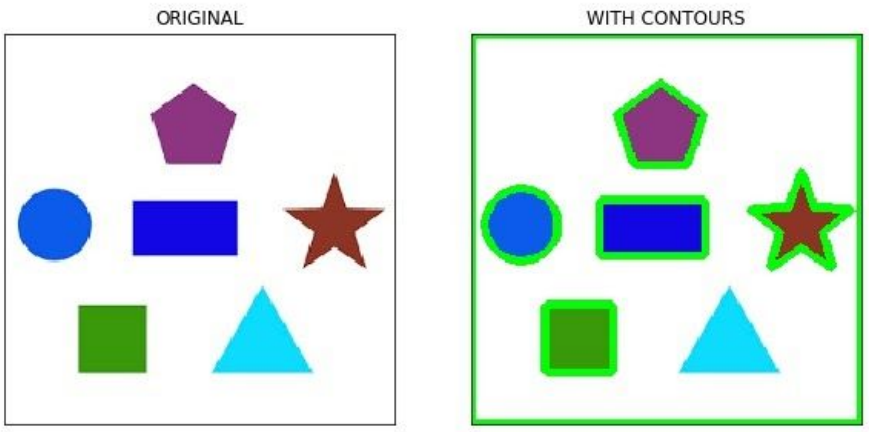
Image Contours

A contour is a closed curve of points or line segments that represents the boundaries of an object in the image. Contours are essentially the shapes of objects in an image.

Unlike edges, contours are not part of an image. Instead, they are an abstract collection of points and line segments corresponding to the shapes of the object(s) in the image.

We can use contours to count the number of objects in an image, categorize objects on the basis of their shapes, or select objects of particular shapes from the image.

```
#importing the required libraries

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline

image = cv2.imread('shapes.png')

#converting RGB image to Binary

gray_image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)

ret,thresh = cv2.threshold(gray_image,127,255,0)

#calculate the contours from binary image

im,contours,hierarchy =
cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)

with_contours = cv2.drawContours(image,contours,-1,(0,255,0),3)

plt.imshow(with_contours)
```



Scale Invariant Feature Transform (SIFT)

Keypoints is a concept you should be aware of when working with images. These are basically the points of interest in an image. Keypoints are analogous to the features of a given image.

They are locations that define what is interesting in the image. Keypoints are important, because no matter how the image is modified (rotation, shrinking, expanding, distortion), we will always find the same keypoints for the image.

Scale Invariant Feature Transform (SIFT) is a very popular keypoint detection algorithm. It consists of the following steps:

- Scale-space extrema detection
- Keypoint localization
- Orientation assignment
- Keypoint descriptor
- Keypoint matching

Features extracted from SIFT can be used for applications like image stitching, object detection, etc. The below code and output show the keypoints and their orientation calculated using SIFT.

```
#import required libraries

import cv2
```

```
import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

#show OpenCV version

print(cv2.__version__)

#read the iamge and convert to grayscale

image = cv2.imread('index.png')

gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)

#create sift object

sift  = cv2.xfeatures2d.SIFT_create()

#calculate keypoints and their orientation

keypoints,descriptors = sift.detectAndCompute(gray,None)

#plot keypoints on the image

with_keypoints = cv2.drawKeypoints(gray,keypoints)

#plot the image

plt.imshow(with_keypoints)
```

Speeded-Up Robust Features (SURF)

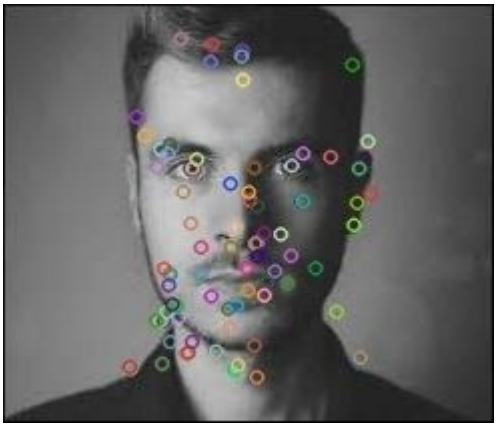
Speeded-Up Robust Features (SURF) is an enhanced version of SIFT. It works much faster and is more robust to image transformations. In SIFT, the scale space is approximated using Laplacian of Gaussian. Wait – that sounds too complex. What is Laplacian of Gaussian?

Laplacian is a kernel used for calculating the edges in an image. The Laplacian kernel works by approximating a second derivative of the image. Hence, it is very sensitive to noise. We generally apply the Gaussian kernel to the image before Laplacian kernel thus giving it the name Laplacian of Gaussian.

In SURF, the Laplacian of Gaussian is calculated using a box filter (kernel). The convolution with box filter can be done in parallel for different scales which is the underlying reason for the enhanced speed of SURF (compared to SIFT). There are other neat improvements like this in SURF – I suggest going through the research paper to understand this in-depth.

```
#import required libraries

import cv2

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline
```

```
#show OpenCV version

print(cv2.__version__)

#read image and convert to grayscale

image = cv2.imread('index.png')

gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)

#instantiate surf object

surf  = cv2.xfeatures2d.SURF_create(400)

#calculate keypoints and their orientation

keypoints,descriptors = surf.detectAndCompute(gray,None)



with_keypoints = cv2.drawKeypoints(gray,keypoints)



plt.imshow(with_keypoints)
```

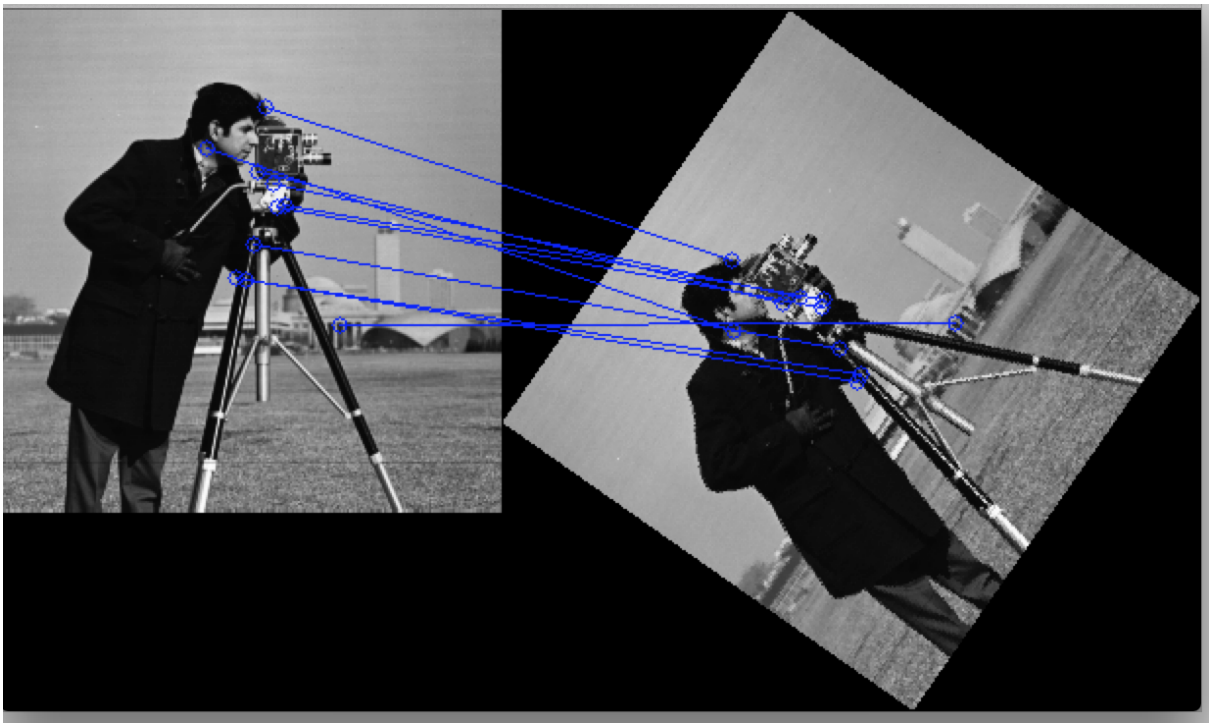


Feature Matching

The features extracted from different images using SIFT or SURF can be matched to find similar objects/patterns present in different images. The OpenCV library supports multiple feature-matching algorithms, like brute force matching, knn feature matching, among others.

```
import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline



#reading images in grayscale format

image1 = cv2.imread('messi.jpg',0)

image2 = cv2.imread('team.jpg',0)



#finding out the keypoints and their descriptors

keypoints1,descriptors1 = cv2.detectAndCompute(image1,None)

keypoints2,descriptors2 = cv2.detectAndCompute(image2,None)
```

```
#matching the descriptors from both the images

bf = cv2.BFMatcher()

matches = bf.knnMatch(ds1,ds2,k = 2)









#selecting only the good features

good_matches = []

for m,n in matches:

   if m.distance < 0.75*n.distance:

       good.append([m])



image3 = cv2.drawMatchesKnn(image1,kp1,image2,kp2,good,flags = 2)
```



In the above image, we can see that the keypoints extracted from the original image (on the left) are matched to keypoints of its rotated version. This is because the features were extracted using SIFT, which is invariant to such transformations.

Face Detection

OpenCV supports haar cascade based object detection. Haar cascades are machine learning based classifiers that calculate different features like edges, lines, etc in the image. Then, these classifiers train using multiple positive and negative samples.

Trained classifiers for different objects like faces,eyes etc are available in the OpenCV Github repo , you can also train your own haar cascade for any object.

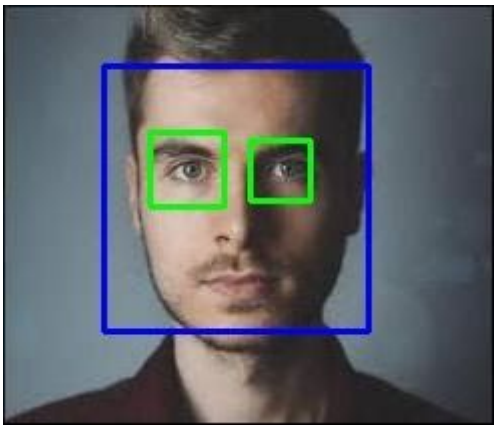
Make sure you go through the below excellent article that teaches you how to build a face detection model from video using OpenCV:

- Building a Face Detection Model from Video using Deep Learning (OpenCV Implementation)

And if you're looking to learn the face detection concept from scratch, then this article should be of interest.

```
#import required libraries

import numpy as np

import cv2 as cv

import matplotlib.pyplot as plt

%matplotlib inline



#load the classifiers downloaded

face_cascade = cv.CascadeClassifier('haarcascade_frontalface_default.xml')

eye_cascade = cv.CascadeClassifier('haarcascade_eye.xml')

#read the image and convert to grayscale format

img = cv.imread('rotated_face.jpg')

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

#calculate coordinates

faces = face_cascade.detectMultiScale(gray, 1.1, 4)

for (x,y,w,h) in faces:

    cv.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_cascade.detectMultiScale(roi_gray)

    #draw bounding boxes around detected features

    for (ex,ey,ew,eh) in eyes:

        cv.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)

#plot the image

plt.imshow(img)

#write image

cv2.imwrite('face_detection.jpg',img)
```



End Notes

OpenCV is truly an all emcompassing library for computer vision tasks. I hope you tried out all the above codes on your machine – the best way to learn computer vision is by applying it on your own. I encourage you to build your own applications and experiment with OpenCV as much as you can.