**The Robotics Forum**

**Robosoft**

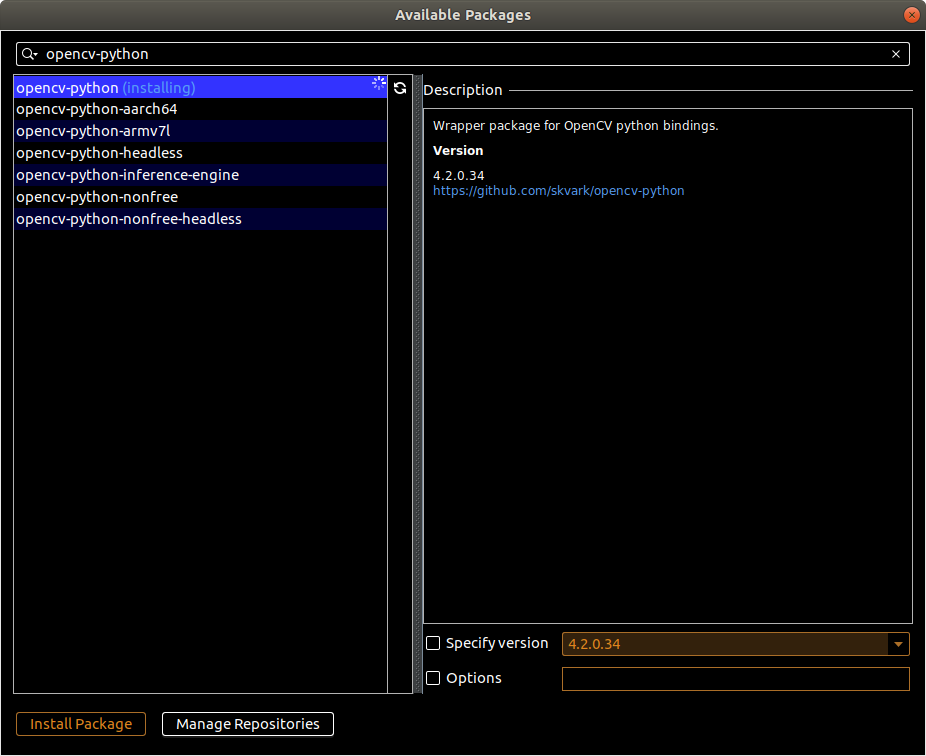
Level 2 workshop

Domain: Image processing

## Prerequisites :

**Opencv-python-** As it’s name suggests, OpenCV is a library of Python bindings. It’s main purpose is to solve Computer vision / Image processing problems. All basic tasks can be done using this library. Throughout our workshop, we are going to use this library. To use it, we should first load it in our project. For that go to **File-->Settings.** Search **‘Project Interpreter’**. A window shown below will appear. 

Check out that there is a ‘**+’** sign on the right hand side. Click on that sign. A new window with a list of modules will appear. Search for **opencv-python** (check the correct spelling!). Now there will be a button, named **Install Package**. Click on it. Your PC should have stable internet connection to install packages.



After successful installation you will see a message as Package ‘opencv-python’ installed successfully. Now we have to import the package in our project. So create a python file from “New” option and write a line below:

*import cv2*

If the package is installed correctly then it should not produce any error.

## 

## 

Read an image

Use the function cv2.imread() to read an image. The image should be in the working directory or full path of image should be given. Second argument is a flag which specifies the way the image should be read.

• cv2.IMREAD\_COLOR : Loads a color image. Any transparency of image will be neglected. It is the default

flag.

• cv2.IMREAD\_GRAYSCALE : Loads image in grayscale mode

• cv2.IMREAD\_UNCHANGED : Loads image as such including alpha channel

Note: Instead of these three flags, you can simply pass integers 1, 0 or -1 respectively.

*import cv2*

*# Load an color image in grayscale*

*img = cv2.imread('messi5.jpg',0)*

**Display an image**

Use the function cv2.imshow() to display an image in a window.

First argument is a window name which is a string. Second argument is our image. You can create as many windows as you wish, but with different window names.

*cv2.imshow('image',img)*

*cv2.waitKey(0)*

*cv2.destroyAllWindows()*

**cv2.waitKey()** is a keyboard binding function. Its argument is the time in milliseconds. The function waits for specified milliseconds for any keyboard event. If you press any key in that time, the program continues. If 0 is passed, it waits indefinitely for a keystroke.

**cv2.destroyAllWindows()** simply destroys all the windows we created. If you want to destroy any specific window, use the function cv2.destroyWindow() where you pass the exact window name as the argument.

**Write an image**

Use the function cv2.imwrite() to save an image.

First argument is the file name, second argument is the image you want to save.

*cv2.imwrite('messigray.png',img)*

This will save the image in PNG format in the working directory.

**Capture Video from Camera**

To capture a video, you need to create a VideoCapture object. Its argument can be either the device index or the name of a video file. Device index is just the number to specify which camera. Normally we use a webcam which is inbuilt. For using it we pass 0 as argument. If we want to use an externally connected camera then the argument should be 1. After that, you can capture frame-by-frame. But at the end, don’t forget to release the capture.

*import cv2*

*cap = cv2.VideoCapture(0) #using webcam*

*while(True):*

*# Capture frame-by-frame*

*ret, frame = cap.read()*

*# Our operations on the frame come here*

*gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)*

*# Display the resulting frame*

*cv2.imshow('frame',gray)*

*if cv2.waitKey(1) & 0xFF == ord('q'):*

*break*

*# When everything done, release the capture*

*cap.release()*

*cv2.destroyAllWindows()*

**cap.read()** returns a bool (True/False). If the frame is read correctly, it will be True. So you can check the end of the video by checking this return value.

Sometimes, cap may not have initialized the capture. In that case, this code shows error. You can check whether it is initialized or not by the method cap.isOpened(). If it is True, OK. Otherwise open it using cap.open().

**Saving a Video**

So we capture a video and process it frame-by-frame, and we want to save that video. For images, it is very simple: just use [cv.imwrite()](https://docs.opencv.org/master/d4/da8/group__imgcodecs.html#gabbc7ef1aa2edfaa87772f1202d67e0ce). Here, a little more work is required.

This time we create a **VideoWriter** object. We should specify the output file name (eg: output.avi). Then we should specify the **FourCC** code (details in next paragraph). Then number of frames per second (fps) and frame size should be passed. And the last one is the **isColor** flag. If it is True, the encoder expects a color frame, otherwise it works with a grayscale frame.

[FourCC](http://en.wikipedia.org/wiki/FourCC) is a 4-byte code used to specify the video codec. The list of available codes can be found in [fourcc.org](http://www.fourcc.org/codecs.php). It is platform dependent. The following codecs work fine for me.

* In Fedora: DIVX, XVID, MJPG, X264, WMV1, WMV2. (XVID is more preferable. MJPG results in high size video. X264 gives very small size video)
* In Windows: DIVX (More to be tested and added)
* In OSX: MJPG (.mp4), DIVX (.avi), X264 (.mkv).

FourCC code is passed as `cv.VideoWriter\_fourcc('M','J','P','G')or cv.VideoWriter\_fourcc(\*'MJPG')` for MJPG.

*import cv2*

*cap = cv2.VideoCapture(0)*

*fourcc = cv2.VideoWriter\_fourcc(\*'XVID')*

*out = cv2.VideoWriter('output.avi',fourcc, 20.0, (640,480))*

*while(cap.isOpened()):*

*ret, frame = cap.read()*

*if ret==True:*

*frame = cv2.flip(frame,0)*

*# write the flipped frame*

*out.write(frame)*

*cv2.imshow('frame',frame)*

*if cv2.waitKey(1) & 0xFF == ord('q'):*

*Break*

*else:*

*break*

*# Release everything if job is finished*

*cap.release()*

*out.release()*

*cv2.destroyAllWindows()*

**Drawing Functions in OpenCV**

**Drawing Line**

To draw a line, you need to pass starting and ending coordinates of the line. We will create a black image and draw a blue line on it from top-left to bottom-right corners.

*import numpy as np*

*import cv2*

*# Create a black image*

*img = np.zeros((512,512,3), np.uint8)*

*# Draw a diagonal blue line with thickness of 5 px*

*img = cv2.line(img,(0,0),(511,511),(255,0,0),5)*

**Drawing Rectangle**

To draw a rectangle, you need the top-left corner and bottom-right corner of the rectangle. This time we will draw a green rectangle at the top-right corner of the image.

*import numpy as np*

*import cv2*

*# Create a black image*

*img = np.zeros((512,512,3), np.uint8)*

*#draw rectangle*

*img = cv2.rectangle(img,(384,0),(510,128),(0,255,0),3)*

**Drawing Circle**

To draw a circle, you need its center coordinates and radius. We will draw a circle inside the rectangle drawn above.

*img = cv2.circle(img,(447,63), 63, (0,0,255), -1)*

**Drawing Ellipse**

To draw the ellipse, we need to pass several arguments. One argument is the center location (x,y). Next argument is axes lengths (major axis length, minor axis length). Angle is the angle of rotation of an ellipse in an anti-clockwise direction. startAngle and endAngle denotes the starting and ending of ellipse arc measured in clockwise direction

from the major axis. i.e. giving values 0 and 360 gives the full ellipse.

*# Below example draws a half ellipse at the center of the image.*

*img = cv2.ellipse(img,(256,256),(100,50),0,0,180,255,-1)*

**Drawing Polygon**

To draw a polygon, first you need coordinates of vertices. Make those points into an array of shape ROWSx1x2 where ROWS are the number of vertices and it should be of type int32. Here we draw a small polygon of with four vertices in

yellow color.

*pts = np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)*

*pts = pts.reshape((-1,1,2))*

*img = cv2.polylines(img,[pts],True,(0,255,255))*

**Adding Text to Images**

There is a function named cv2.putText which takes following arguments:

* **img** – Image.
* **text** – Text string to be drawn.
* **org** – Bottom-left corner of the text string in the image.
* **font** – CvFont structure.
* **fontFace** – Font type.
* **fontScale** – Font scale factor that is multiplied by the font-specific base size.
* **color** – Text color.
* **thickness** – Thickness of the lines used to draw a text.
* **lineType** – Line type.

*font = cv2.FONT\_HERSHEY\_SIMPLEX*

*cv2.putText(img,'OpenCV',(10,500), font, 4,(255,255,255),2,cv2.LINE\_AA)*

**Image Blending**

This is also image addition, but different weights are given to images so that it gives a feeling of blending or transparency.

Images are added as per the equation below:

𝑔(𝑥) = (1 − 𝛼)𝑓0(𝑥) + 𝛼𝑓1(𝑥)

By varying 𝛼 from 0 → 1, you can perform a cool transition between one image to another.

Here I took two images to blend them together. First image is given a weight of 0.7 and the second image is given 0.3.

cv2.addWeighted() applies the following equation on the image.

𝑑𝑠𝑡 = 𝛼 · 𝑖𝑚𝑔1 + 𝛽 · 𝑖𝑚𝑔2 + 𝛾

Here 𝛾 is taken as zero.

*img1 = cv2.imread('ml.png')*

*img2 = cv2.imread('opencv\_logo.jpg')*

*dst = cv2.addWeighted(img1,0.7,img2,0.3,0)*

*cv2.imshow('dst',dst)*

*cv2.waitKey(0)*

*cv2.destroyAllWindows()*

**Image Thresholding**

Simple Thresholding

Here, the matter is straight forward. If a pixel value is greater than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black). The function used is cv2.threshold. First argument is the source image, which should be a grayscale image. Second argument is the threshold value which is used to classify

the pixel values. Third argument is the maxVal which represents the value to be given if pixel value is more than (sometimes less than) the threshold value. OpenCV provides different styles of thresholding and it is decided by the

fourth parameter of the function. Different types are:

• cv2.THRESH\_BINARY

• cv2.THRESH\_BINARY\_INV

• cv2.THRESH\_TRUNC

• cv2.THRESH\_TOZERO

• cv2.THRESH\_TOZERO\_INV

Documentation clearly explain what each type is meant for. Please check out the documentation.

Two outputs are obtained. First one is a retval which will be explained later. Second output is our thresholded image.

*import cv2*

*import numpy as np*

*from matplotlib import pyplot as plt*

*img = cv2.imread('gradient.png',0)*

*ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH\_BINARY)*

*ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH\_BINARY\_INV)*

*ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH\_TRUNC)*

*ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH\_TOZERO)*

*ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH\_TOZERO\_INV)*

*titles = ['Original Image','BINARY','BINARY\_INV','TRUNC','TOZERO','TOZERO\_INV']*

*images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]*

*for i in xrange(6):*

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

*plt.title(titles[i])*

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

*plt.show()*

Adaptive Thresholding

In the previous section, we used a global value as threshold value. But it may not be good in all the conditions where the image has different lighting conditions in different areas. In that case, we go for adaptive thresholding. In this, the algorithm calculates the threshold for a small region of the image. So we get different thresholds for different regions of the same image and it gives us better results for images with varying illumination.

It has three ‘special’ input params and only one output argument.

Adaptive Method - It decides how thresholding value is calculated.

• cv2.ADAPTIVE\_THRESH\_MEAN\_C : threshold value is the mean of neighbourhood area.

• cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C : threshold value is the weighted sum of neighbourhood values where weights are a gaussian window.

Block Size - It decides the size of a neighbourhood area.

C - It is just a constant which is subtracted from the mean or weighted mean calculated.

Below piece of code compares global thresholding and adaptive thresholding for an image with varying illumination:

*import cv2*

*import numpy as np*

*from matplotlib import pyplot as plt*

*img = cv2.imread('dave.jpg',0)*

*img = cv2.medianBlur(img,5)*

*ret,th1 = cv2.threshold(img,127,255,cv2.THRESH\_BINARY)*

*th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE\_THRESH\_MEAN\_C,\*

*cv2.THRESH\_BINARY,11,2)*

*th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,\*

*cv2.THRESH\_BINARY,11,2)*

*titles = ['Original Image', 'Global Thresholding (v = 127)',*

*'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']*

*images = [img, th1, th2, th3]*

*for i in xrange(4):*

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

*plt.title(titles[i])*

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

*plt.show()*

**Image Blurring (Image Smoothing)**

**Averaging**

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

A 3x3 normalized box filter would look like this:

K = (1/9) \* ⎡ 1 1 1

| 1 1 1

1 1 1 ⎦

*import cv2*

*import numpy as np*

*from matplotlib import pyplot as plt*

*img = cv2.imread('opencv\_logo.png')*

*blur = cv2.blur(img,(5,5))*

*plt.subplot(121),plt.imshow(img),plt.title('Original')*

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

*plt.subplot(122),plt.imshow(blur),plt.title('Blurred')*

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

*plt.show()*

**Gaussian Filtering**

In this approach, instead of a box filter consisting of equal filter coefficients, a Gaussian kernel is used. It is done with the function, cv2.GaussianBlur(). We should specify the width and height of the kernel which should be positive and odd. We also should specify the standard deviation in the X and Y directions, sigmaX and sigmaY respectively. If

only sigmaX is specified, sigmaY is taken as equal to sigmaX. If both are given as zeros, they are calculated from the kernel size. Gaussian filtering is highly effective in removing Gaussian noise from the image.

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

The above code can be modified for Gaussian blurring:

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

**Median Filtering**

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

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

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

**Morphological Transformations**

**Erosion**

The basic idea of erosion is just like soil erosion only, it erodes away the boundaries of foreground object (Always try to keep foreground in white). So what does it do? The kernel slides through the image (as in 2D convolution). A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero).So what happens is that, all the pixels near boundary will be discarded depending upon the size of kernel. So the thickness or size of the foreground object decreases or simply white region decreases in the image. It is useful for removing small white noises (as we have seen in colorspace chapter), detach two connected objects etc.

Here, as an example, I would use a 5x5 kernel with full of ones. Let’s see it how it works:

*import cv2*

*import numpy as np*

*img = cv2.imread('j.png',0)*

*kernel = np.ones((5,5),np.uint8)*

*erosion = cv2.erode(img,kernel,iterations = 1)*

**Dilation**

It is just opposite of erosion. Here, a pixel element is ‘1’ if atleast one pixel under the kernel is ‘1’. So it increases the white region in the image or size of foreground object increases. Normally, in cases like noise removal, erosion is followed by dilation. Because, erosion removes white noises, but it also shrinks our object. So we dilate it. Since noise is gone, they won’t come back, but our object area increases. It is also useful in joining broken parts of an object.

*dilation = cv2.dilate(img,kernel,iterations = 1)*

**Opening**

Opening is just another name of erosion followed by dilation. It is useful in removing noise, as we explained above. Here we use the function, cv2.morphologyEx()

*opening = cv2.morphologyEx(img, cv2.MORPH\_OPEN, kernel)*

**Closing**

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

*closing = cv2.morphologyEx(img, cv2.MORPH\_CLOSE, kernel)*

**Morphological Gradient**

It is the difference between dilation and erosion of an image.

The result will look like the outline of the object.

*gradient = cv2.morphologyEx(img, cv2.MORPH\_GRADIENT, kernel)*

**Motion Detection**

Detecting motion is a very simple task. We can solve it by this basic idea that if an object in the first frame is at position X and in the second frame, its at X+1, then there is motion of 1. So basically we can detect the motion in two consecutive frame by just taking the difference in the position of pixels. We use cv2.absdiff() for this difference. It takes new frame and old frame as argument.

*import cv2*

*import numpy as np*

*cap=cv2.VideoCapture(0)*

*ret,frame1 = cap.read(0)*

*old = cv2.cvtColor(frame1,cv2.COLOR\_BGR2GRAY)*

*while True:*

*ret,frame = cap.read(0) # here also 0 is use for same*

*new=cv2.cvtColor(frame,cv2.COLOR\_BGR2GRAY)*

*diff=cv2.absdiff(new,old)*

*cv2.imshow('frame12',diff)*

*old=new*

*if cv2.waitKey(1)==32:*

*break*

*cap.release()*

*cv2.destroyAllWindows()*

**Masking**

We know that colors are denoted by numbers in Computer vision. So masking is just like replacing some numbers in a specific range by a value. Here we are going to create trackbars, which will select the range to which we want to mask. Color masking has many applications in color related tasks.

*import numpy as np*

*import cv2*

*cap=cv2.VideoCapture(0)*

*def nothing(x):*

*pass*

*def colormask():*

*cv2.namedWindow("Masking",1)*

*#set trackbar*

*bh='blue high'*

*bl='blue low'*

*gh='green high'*

*gl='green low'*

*rh='red high'*

*rl='red low'*

*#set range*

*cv2.createTrackbar(bl,"Masking",0,255,nothing)*

*cv2.createTrackbar(gl,"Masking",0,255,nothing)*

*cv2.createTrackbar(rl,"Masking",0,255,nothing)*

*cv2.createTrackbar(bh,"Masking",0,255,nothing)*

*cv2.createTrackbar(gh,"Masking",0,255,nothing)*

*cv2.createTrackbar(rh,"Masking",0,255,nothing)*

*while True:*

*ret,frame=cap.read(0)*

*#convert rgb to hsv*

*hsv=cv2.cvtColor(frame,cv2.COLOR\_BGR2HSV)*

*btl=cv2.getTrackbarPos(bl,"Masking")*

*gtl=cv2.getTrackbarPos(gl,"Masking")*

*rtl=cv2.getTrackbarPos(rl,"Masking")*

*bth=cv2.getTrackbarPos(bh,"Masking")*

*gth=cv2.getTrackbarPos(gh,"Masking")*

*rth=cv2.getTrackbarPos(rh,"Masking")*

*rgbl=np.array([btl,gtl,rtl],np.uint8)*

*rgbh=np.array([bth,gth,rth],np.uint8)*

*mask=cv2.inRange(hsv,rgbl,rgbh)*

*cv2.imshow("Mask",mask)*

*cv2.imshow("ori",frame)*

*k=cv2.waitKey(1)*

*if k==32:*

*break*

*colormask()*

*cap.release()*

*cv2.destroyAllWindows()*

**Face Detection**

Face detection is one of the coolest task of IP. In this model the Haar Cascade is trained to detect human face. To train it, 1000s of positive and negative images are given. But here we will be using pre trained .xml file which will detect face.

*import cv2*

*import numpy as np*

*face\_cascade =cv2.CascadeClassifier('haarcascade\_frontalface\_default.xml')*

*cap=cv2.VideoCapture(0)*

*while True:*

*rat,frame=cap.read(0)*

*gray=cv2.cvtColor(frame,cv2.COLOR\_BGR2GRAY)*

*faces=face\_cascade.detectMultiScale(gray,1.3,4)*

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

*cv2.rectangle(frame,(x,y),(x+w,y+h),(255,0,0),2)*

*cv2.imshow('img',frame)*

*k=cv2.waitKey(1)*

*if k==32:*

*break*

*cap.release()*

*cap.destroyAllWindows()*