TRS Winter School – Computer Vision Vertical – Documentation

Pre-requisite Packages:

- Numpy: `pip install numpy`
- OpenCV (cv2) : `pip install opencv-python`
- Serial (For working with Arduino): `pip install serial`

Basics of Numpy

```
import numpy as np                  # to import the numpy package
(imported as "np" for simplicity in typing)

a = np.array([[1,2,3],[4,5,6]])   # creates an array (rank = 2)
n,m = a.shape                       # returns shape of the array
(rows,columns)

b = np.zeros((n,m)                  # fills an (m,n) array with zeros
c = np.full((n,m),x)                # fills an (m,n) array with "x"
```

Basics of OpenCV (cv2)

Displaying an Image

```
import cv2                                      # importing the library
img = cv2.imread('image.png')                   # reading the image (here it
is image.png)

cv2.namedWindow('Window',cv2.WINDOW_NORMAL) # creating a named window
cv2.imshow('Window',img)                        # showing the image in this
window

cv2.waitKey(0)                                  # waits for user input after
the no. of        milliseconds specified. If '0', it keeps checking for
input
cv2.destroyAllWindows()                         # destroys all open windows
```

Displaying a Matrix as an Image (With example of a circle)

- In the case of **grayscale images**, **0 value in matrix represents black** and 255 represents white, and different intensities are present between 0 to 255.
- In this case we create an array of type (m,n).
- In the case of **color images**, we need to create a **3-dimensional array** with a value between 0-255 for each color (in the order B, G, R), **0 representing minimum intensity** and **255 representing maximum intensity** of that color.
- In this case we create an array of type (m,n,3).
- While displaying a matrix as an image, **we need to convert it to type uint8**, which is the one compatible with OpenCV.

```
import cv2
import numpy as np
import math
matrix = np.full((800,800),255) # making a completely white image first

# looping through the pixels of the image
for i in range(matrix.shape[0]):
    for j in range(matrix.shape[1]):
        # (400,400) is centre, 400 is radius
```

```
        # we will use eqn of circle to check if pixel needs to be
colored black
        if(math.sqrt((400-i)**2+(400-j)**2)<=200):
            matrix[i,j] = 0

cv2.namedWindow('Circle',cv2.WINDOW_NORMAL)
cv2.imshow('Circle',matrix.astype(np.uint8))

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Creating a Trackbar

- A trackbar can be created using the `cv2.createTrackbar()` function.
- The parameters required are: Trackbar name, Window name, Default value, Maximum value, and function to be executed (with argument as trackbar value)
- Here is an example:

```
"""Creating a trackbar to control brightness of image"""
import cv2
import numpy as np

add_value = 25 # default value by which brightness is incremented

img = cv2.imread('grayDog.jpeg',cv2.IMREAD_GRAYSCALE) # reading already
present image

def incBrightness(value):
    # The trackbar value becomes the value by which brightness is
incremented
    global add_value
    add_value = value

cv2.namedWindow('Image', cv2.WINDOW_NORMAL)
cv2.imshow('Image',img)
```

```
cv2.createTrackbar('Brightness','Image',50,255,incBrightness)

while(1):
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if img[i,j] < 250 - add_value:
                img[i,j] += add_value
            else:
                img[i,j] = 255
    cv2.imshow('Image',img.astype(np.uint8))

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Exploring some Inbuilt Functions of OpenCV Fundamentally

Converting RGB Image to Grayscale Image

Elementary Method: 3 Ways

1. Taking average of B, G, R values:

```
matrix[i,j] = 0.33*b[i,j] + 0.33*g[i,j] + 0.33*r[i,j]
```

2. Since human eye can perceive green color the most, we give it the most weightage:

```
matrix[i,j] = 0.114*b[i,j] + 0.587*g[i,j] + 0.299*r[i,j]
```

   OR

```
matrix[i,j] = 0.07*b[i,j] + 0.72*g[i,j] + 0.21*r[i,j]
```

3. Taking sum of greatest and lowest out of the B, G, R values:

```
matrix[i,j] = 0.5*max(r[i,j],g[i,j],b[i,j]) +
0.5*min(r[i,j],g[i,j],b[i,j])
```

Inbuilt Function:

- Using OpenCV's inbuilt function `cvtColor()` with flag `COLOR_BGR2GRAY`

```
img = cv2.imread('image.png')
matrix = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```

Translating an Image

Elementary Method:

- Creating a new blank matrix of the same dimensions, and transferring the pixels by applying the translation

```
# here tx and ty are the translations in x and y respectively
if(i>tx and j>ty): #ensuring that it doesnt go out of bounds
  new_matrix[i,j] = img[i-tx,j-ty]
```

Inbuilt Function:

- Using OpenCV's inbuilt function `warpAffine()`

```
# we first create an array that represents translation in x and y
matrix = np.array([[1,0,tx],[0,1,ty]],dtype=np.float32)
new = cv2.warpAffine(src=img,M=matrix,dsize=(n,m))
```

Rotating an Image

Elementary Method:

- Creating a new blank matrix of a larger dimension (to avoid going out of bounds) and shifting each pixel using rotation formula

```
# here theta is the angle by which image needs to be rotated (in
radians)
s = math.sin(-theta)
c = math.cos(-theta)
new = np.full((1000,1000,3),0)
# We move from the centre to left and right
for i in range(int(-n/2),int(n/2)):
    for j in range(int(-m/2),int(m/2)):
        new[500+int(i*c+j*s),500+int(j*c-i*s),0] =
img[int(n/2)+i,int(m/2)+j,0]
        new[500+int(i*c+j*s),500+int(j*c-i*s),1] =
img[int(n/2)+i,int(m/2)+j,1]
        new[500+int(i*c+j*s),500+int(j*c-i*s),2] =
img[int(n/2)+i,int(m/2)+j,2]
```

Inbuilt Function:

- Using OpenCV's inbuilt functions `getRotationMatrix2D()` and `warpAffine()`

```
# here the angle theta is in degrees NOT radians
center = (n/2,m/2)
rotated_matrix =
cv2.getRotationMatrix2D(center=center,angle=theta,scale=1)
rotated_img = cv2.warpAffine(src=img,M=rotated_matrix,dsize=(n,m))
```

Padding an Image

Elementary Method:

- Creating a new larger matrix with color as that required of padding, and transfering the pixels while leaving space at the borders.

```
n+=40
m+=40
matrix = np.full((n,m,3),0)
for i in range(n):
     for j in range(m):
         if (i>20 and i<n-20) and (j>20 and j<m-20):
            matrix[i,j,0] = img[i-20,j-20,0]
            matrix[i,j,1] = img[i-20,j-20,1]
            matrix[i,j,2] = img[i-20,j-20,2]
```

Inbuilt Function:

- Using OpenCV's inbuilt function `copyMakeBorder()`

```
matrix = cv2.copyMakeBorder(img,20,20,20,20,cv2.BORDER_NORMAL,None,
(255,0,0))
```

- After the parameter `img`, **the next four parameters are the distances of the border from top, bottom, left and right respectively**
- The different border types are:
  - **BORDER_NORMAL:** Constant colored border -> Color will be in (B,G,R) form (last parameter)
  - **BORDER_REFLECT:** Border will be mirror reflection of the image
  - **BORDER_REFLECT_101 or BORDER_DEFAULT:** Slightly different from BORDER_REFLECT
  - **BORDER_REPLICATE:** Replicates the last element

Resizing an Image

Elementary Method:

1. Upsizing

- Create a new matrix of dimensions that will be more than the original dimensions by the factor of the scale
- Loop through the original image and fill a corresponding square of size scale*scale in the new matrix with the color of the original image

```
# here 'scale' is the scale by which it is upsized
matrix = np.full((n*scale,m*scale,3),255)
for i in range(img.shape[0]):
  for j in range(img.shape[1]):
    for a in range(scale*i, scale*(i+1)):
      for b in range(scale*j, scale*(j+1)):
        matrix[a,b] = img[i,j]
```

2. Downsizing

- Create a new matrix of dimensions that will be less than the original dimensions by the factor of the scale (it should be rounded to an integer)
- Loop through the original image and for each square of size scale*scale, calculate the average of the B, G, R values and fill the corresponding pixel of new matrix with this average

```
new_n, new_m = int(n/scale), int (m/scale)
matrix = np.full((new_n,new_m,l),0)
for i in range(new_n):
  for j in range(new_m):
    sum_b,sum_g,sum_r=0,0,0
    for a in range(scale*i, scale*(i+1)):
      for b in range(scale*j, scale*(j+1)):
        sum_b += img[a,b,0]
        sum_g += img[a,b,1]
```

```
            sum_r += img[a,b,2]
            sum_b/=(scale**2)
            sum_g/=(scale**2)
            sum_r/=(scale**2)
            matrix[i,j,0]=sum_b/(scale)
            matrix[i,j,1]=sum_g/(scale)
            matrix[i,j,2]=sum_r/(scale)
```

Inbuilt Function:

- Using OpenCV's inbuilt function `resize()` (for both upscaling and downscaling)
- Here it is to be noted that the dimensions are taken in reverse order, i.e., (columns,rows) instead of (rows, columns)

```
# the new scaled dimensions of the image are 'n_new' and 'm_new'
matrix = cv2.resize(img,(m_new,n_new),interpolation = cv2.INTER_AREA)
```

Reversing an Image

Elementary Method:

- Creating a new matrix and transfering the pixels from original image in reverse order

```
for i in range(n):
    for j in range(m):
        matrixX[i,j] = imgMatrix[n-i-1,j] # reversed about X
        matrixY[i,j] = imgMatrix[i,m-j-1] # reversed about Y
        matrixXY[i,j] = imgMatrix[n-i-1,m-j-1] # reversed about X and Y
```

Inbuilt Function:

- Using OpenCV's inbuilt function `flip()`

```
matrix = cv2.flip(img,flipCode) # flipCode: 0=>Vertical, 1=>Horizontal,
-1=>Both
```

Annotating an Image

- We can annotate an image with a line or a circle.

```
# annotating a line:
ptA = (x1,y1)
ptB = (x2,y2)
# draws a line through the image from ptA to ptB
cv2.line(imgLine,ptA,ptB,color,thickness,lineType)
```

```
# annotating a circle:
cv2.circle(img,center,radius,color,thickness,lineType)
# Giving thickness as -1 makes it a filled circle
```

Erosion of an Image

- Done to remove noise from image

```
# 'kernel' stores the dimensions of the matrix that will be used as
kernel
cv2.erode(img,kernel)
```

Dilation of an image

- Done to increase white regions in image

```
# 'kernel' stores the dimensions of the matrix that will be used as
kernel
cv2.dilate(img,kernel)
```

Thresholding of a Greyscale Image

- This is done to obtain a perfect binary image
- If a **pixel has value <127**, it is **converted to 0**.
- Similarly, if a **pixel has value >127**, it is **converted to 255**.

```
for i in range(img.shape[0]):
  for j in range(img.shape[1]):
    if (img[i,j]<127):
      img[i,j] = 0
    else:
      img[i,j] = 255
```

HSV (Hue-Saturation-Value) Color Space

- HSV is used more often in CV because of its superior performance in varying illumination levels as compared to RGB. It helps us do better segmentation and filtering on objects even under varying resolution and lighting conditions.
- Hence thresholding and masking (covered later) is usually done in HSV color space.
- Hue is the color which lies in [0,179]
- Saturation is the greyness which lies in [0,255]
- Value is the brightness which lies in [0,255]

Conversion from BGR to HSV

- We use OpenCV's inbuilt function `cvtColor()` with flag `COLOR_BGR2HSV`

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Edge Detection Methods

1. Sobel Filter

- Used to get gradient in each direction.

```
cv2.Sobel(img, ddepth, dx, dy, ksize)
```

- **dx and dy:** direction in which you want to take gradient
- **ksize:** size of filter (kernel) which could be 3,5,7
- **ddepth:** taken as CV_64F

2. Canny Edge Detection

- **Involves 4 steps of processing** rather than just finding intensity gradient:
  1. Noise Reduction (5x5 Gaussian Blur)
  2. Finding Intensity Gradient of Image (Sobel)
  3. Non-Maximum Suppression
  4. Hysteresis Thresholding

```
cv2.Canny(img,maxVal,minVal)
```

- **maxVal and minVal** are values used in Hysteresis Thresholding; generally taken in ratio 2:1 or 3:1

3. Hough Line Transformation:

- Used to detect curves in **binary** images after they have undergone edge detection (edges depicted as white).
- It works by analysing each pixel in the image and plotting them in a (theta,r) space (Hough plane) if they are white.
- The number of intersecton points gives number of unique lines and coordinates of each

such point guves parameters of each of those lines

```
cv2.HoughLines(img,rho,theta,threshold)
```

- **img:** 8bit, single-channel binary source image
- **rho:** resolution of parameter r
- **theta:** resolution of parameter theta (in radians)
- **threshold:** minimum number of intersections to "detect" a line
- **Read docs for more information**

4. Contours and Contour Hierarchy:

- **Contour:**
  - A **curve joining all the continuous points (along a boundary), having same color/intensity**.
  - It is also found on **binary** image.
  - Before finding countours, **thresholding/canny edge detection** is done.
  - Finding contours is like finding white object from black background, so that colors should be maintained.
  - Using `cv2.findContours()` you can get a list of all the contours and their hierarchy.
  - Then you can use `cv2.drawContours()` to draw those.
  - **Read docs for more information**
- **Contour Hierarchy:**
  - **If a contour is completely within the other**, it is a **child**.
  - **Otherwise**, it is a **sibling**.

Template Matching

- A **method for searching and finding the location of a template image in a larger image.**

```
cv2.matchTemplate(image, template, method)
```

- Different methods:
  - **TM_SQDIFF:** Squared difference between template and image
  - **TM_SQDIFF_NORMED:** Normalise the output of SQDIFF
  - **TM_CCORR:** Multiply the two images together (Cross Correlation). If they line up exactly, they will give highest value. Used generally for signal processing (resonance).
  - **TM_CCORR_NORMED**: Normalise the output of CCORR
  - **TM_CCOEFF:** Subtracts mean of template and image from themselves before comparing to give better results.
  - **TM_CCOEFF_NORMED:** Normalise the output of CCOEFF
- It returns a grayscale image, where each pixel denotes how much the neighbourhood of that pixel matches with template.
- To find maximum score:

```
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc()
```

- **min and max loc** are x and y coordinates of top-left corner
- min and max val are the length and breadth respectively

Detecting Multiple Objects

- Use matchTemplate to give grayscale image.
- Use thresholding to get best output.
- **Limitations:** This method **preserves the orientation of the image**, so it **won't work if the image is rotated**.

Corner Detection

- We check the **intensity** and if there is large intensity variation in all directions, it is a corner.

1. Harris Corner Detection

```
cv2.cornerHarris(img,blocksize,ksize,k)
```

- **blockSize:** neighbourhood size
- **ksize:** size of filter (kernel), **usually taken as 3**.
- **K:** Harris detector free parameter, **usually taken as 0.04**.
- This **returns a grayscale image with "R" values at each point**.
- This image is **thresholded (0.01 * max)** to get the corner points.

Masking

```
# HSV thresholding done for better result
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

# define range of color required in HSV, which can be different
according to need
lower,upper = np.array([30,150,50]),np.array([255,255,100])

# threshold the HSV image with this range
mask = cv2.inRange(hsv, lower, upper) # returns white for all areas that
lies in this range, and black for those that lie outside

res = cv2.bitwise_and(img,img,mask=mask) # we do bitwise AND of mask
with image
```

Working with Videos in OpenCV

```
import cv2
import numpy as np

vid = cv2.VideoCapture('video.mp4') # if we give '0' instead of video
path, it captures video from our webcame

while 1:
    ret, frame = vid.read()         #ret is True or False, denoting
whether it was successful
    cv2.imshow('frame', frame)

    # cv2.waitKey returns a 32 bit binary number which corresponds to
pressed key
    # & (bitwise operator) extracts last 8 bits, and does & with
11111111 (0xFF)

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

vid.release()
cv2.destroyAllWindows()
```

- Here is an example of canny edge detection on a video:

```
import cv2
import numpy as np

vid = cv2.VideoCapture('video.mp4')

while 1:
    ret, frame = vid.read()
    cv2.imshow('frame', frame)
```

```
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        lower,upper = np.array([30,150,50]),np.array([255,255,100])
        mask = cv2.inRange(hsv, lower, upper)
        res = cv2.bitwise_and(frame,frame,mask=mask)
        edges = cv2.Canny(res, 100, 200)
        cv2.imshow('edges', edges)

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

    vid.release()
    cv2.destroyAllWindows()
```

Path-Finding Algorithms

1. Breadth-First Search (BFS) Algorithm

- We traverse all the siblings first, then the children.
- The data structure used is **QUEUE** (First In First Out)
- We go to first line, add all the elements to queue.
- Then we traverse the children of the top-most element of the queue while adding them to the queue, then pop it.
- This is done until the queue is empty.

Example:

```
      A
   B     C
 D     E   F
```

- A first, then B, C, then pop A.
- Then from B we traverse D, and pop B.
- Then from C to E and F, and pop C.
- Then pop D, then E, then F directly as they do not have any children.

- Now the queue is empty
- **Order in queue:** FEDCBA (A in the front)

---

2. Depth-First Search (DFS) Algorithm

- We traverse all the children first, then the siblings
- The data structure used is **STACK**. (Last in First Out)
- We go to first line, add all the elements to stack.
- Then we pop the stack and traverse that element's children.
- This is done until the stack is empty.

Example:

```
      A
   B     C
 D     E   F
```

- A first, then B,C.
- Then pop C and traverse E and F.
- Then pop F, since it has no siblings, we pop E, and again it has no siblings.
- Then pop B and traverse D, then pop D.
- Now the stack is empty.
- **Order in stack:** DFECBA (A in the front)

For shortest path traversal, we go for BFS over DFS

---

3. Dijkstra's Algorithm

- We use this method in connected graphs, where certain elements are connected to each other.
- There is a cost for traversing from one element to another.
- We try to find a path such that the cost is minimized.

- We set the distance of each pixel from the start as infinity, but if it is connected to the start pixel, the distance becomes equal to the cost.
- In the next iteration, we find the element with shortest distance, and from there check if distance of any of the elements can be modified, based on the following formula outlined in pseudo-code:

```
if distance to connected element > ( distance to current element +
cost from current to connected) :
  update distance to connected as distance to current + cost
```

- Similarly we keep iterating until we reach the end.

---

4. A* Algorithm

- This algorithm is **very similar to Dijkstra's Algorithm**.
- The salient difference is that, we **also factor in a heuristic function** (usually the distance from the current element to the end element) while checking for the element with the shortest distance.
- This results in a more guided and hence faster algorithm.

- **For image traversal**, we **consider the image as a tree**, wherein the **adjacent pixels to one pixel will be considered its children**, and **each children pixels are siblings of each other**.

Blob Detection

- **Blob: Binary Large Object**, which is basically a **set of adjacent pixels having the same color**.
- **To find blobs:** iterate through each pixel, once you find one pixel of interested color, then check its neighbourhood, and when you find another pixel of same color, check neighbourhood of that, and so on.
- **Thresholding** is done before finding blobs for more efficiency.

Blob Detection Using BFS and DFS algorithms:

```
import cv2
import numpy as np

img1 = cv2.imread('blob.png',0)
#Blurring it first
img1 = cv2.blur(img1,(5,5)) #(5,5) is kernel size
n,m = img1.shape
img2 = img1.copy() # to perform BFS


# then applying thresholding:
for i in range(img1.shape[0]):
    for j in range(img1.shape[1]):
        if img1[i,j]>127:
            img1 [i,j] = 255
        else:
            img1[i,j] = 0
for i in range(img2.shape[0]):
    for j in range(img2.shape[1]):
        if img2[i,j]>127:
            img2 [i,j] = 255
        else:
            img2[i,j] = 0

def checkDFS(x,y):
    img1[x,y] = 127 #to imply we have visited that pixel
    # checking every child of (x,y)
    if x>0 and img1[x-1,y]==255:
        checkDFS(x-1,y)
    if x<n-1 and img1[x+1,y]==255:
        checkDFS(x+1,y)
    if y>0 and img1[x,y-1]==255:
        checkDFS(x,y-1)
    if y<m-1 and img1[x,y+1]==255:
```

```
            checkDFS(x,y+1)
        if x>0 and y>0 and img1[x-1,y-1]==255:
            checkDFS(x-1,y-1)
        if x>0 and y<m-1 and img1[x-1,y+1]==255:
            checkDFS(x-1,y+1)
        if x<n-1 and y>0 and img1[x+1,y-1]==255:
            checkDFS(x+1,y-1)
        if x<n-1 and y<m-1 and img1[x+1,y+1]==255:
            checkDFS(x+1,y+1)

    queue = []
    def checkBFS(x,y):
        global queue
        img2[x,y] = 127 #to imply we have visited that pixel
        #checking every child of (x,y)
        if x>0 and img2[x-1,y]==255:
            queue.append((x-1,y))
        if x<n-1 and img2[x+1,y]==255:
            queue.append((x+1,y))
        if y>0 and img2[x,y-1]==255:
            queue.append((x,y-1))
        if y<m-1 and img2[x,y+1]==255:
            queue.append((x,y+1))
        if x>0 and y>0 and img2[x-1,y-1]==255:
            queue.append((x-1,y-1))
        if x>0 and y<m-1 and img2[x-1,y+1]==255:
            queue.append((x-1,y+1))
        if x<n-1 and y>0 and img2[x+1,y-1]==255:
            queue.append((x+1,y-1))
        if x<n-1 and y<m-1 and img2[x+1,y+1]==255:
            queue.append((x+1,y+1))

    countDFS = 0
    countBFS = 0
    for i in range(img1.shape[0]):
        for j in range(img1.shape[1]):
```

```
            if img1[i,j]==255:
                countDFS+=1 # keeps track of number of blobs
                checkDFS(i,j)
    for i in range(img1.shape[0]):
        for j in range(img1.shape[1]):
            if img2[i,j]==255:
                countBFS+=1
                checkBFS(i,j)
                while(len(queue)!=0):
                    x,y = queue.pop(0)
                    if(img2[x,y]==255):
                        checkBFS(x,y)


    print(countDFS,countBFS)
```

Implementation of Path-Finding Algorithms

- This was done in Task 1 of the Winter School Team Project, which can be found here: https://github.com/C-12-14/WinterSchool-Team-Project

---

Working with Arduino in Python

- We use the `serial` library to work with Arduino in the Python programming language, as opposed to C++ which is the language usually used for Arduino programming
- **Here is a simple sample code:**

```
    import serial
    import time

    arduino = serial.Serial(port='COM5', baudrate=9600, timeout=.1)

    def write_read(x):
        arduino.write(bytes(x, 'utf-8'))
```

```
        time.sleep(0.05)
        value = arduino.readLine()
        return value

    if __name__ == '__main__':
        while 1:
            num = input("Enter a number: ")
            val = read_write(num)
            print(val)
```

- in `Serial()` function:
  - **port:** Port of the Arduino microcontroller, can be found via Arduino IDE
  - **baudrate:** The speed at which data is transfered between the two communicating systems (PC and Arduino). It is the equivalent of the number entered in `Serial.begin()` in C++.
  - **timeout:** The timeout value. It is the equivalent of `Serial.setTimeout()` in C++.
- By using python to work with Arduino, it opens up a lot of possibilities for using python libraries for Arduino, including OpenCV.
- This could be used in making projects like a self-driving car, which was the intention of task 2 of the Winter School Team Project.