Image Processing – Session 1
OpenCV on a Raspberry Pi

Introduction

In this session, you will be introduced to `NumPy` , `OpenCV` and the PiCamera on a raspberryPi 3B+ or 4. The processes will take 4 steps through which you will manipulate different aspects of the image processing on a Pi. Try not to rush the session and just copy-paste the examples. It is important to understand each line of the codes.
Before starting, make sure that your pi is correctly connected to the Ethernet or the WiFi in case you need to download new packages. Second, run Python in a terminal (e.g. LXTerminal) and test the version of `OpenCV` . To perform the test, execute the following lines in the terminal:

```
$ python3
        >>> import cv2
        >>> print cv2.__version__
        4.6.0
```

Do not try to upgrade Raspbian to `Bullseye` (Debian 11), the support for the packages we will be using completely shifted[1] and is still, at the time of writing, not compatible with the present labs. We stay on `Buster` (Debian 10) for now.
Most of the material of the labs are based on the book "*Raspberry Pi Computer Vision Programming*" by Ashwin Pajankar and published by Packt. Some extra material and codes are available on their GitHub repository[2].

Step 1 – `NumPy` on Raspberry

`NumPy` is a fundamental package that can be used to scientifically compute with Python. `NumPy` comes with many inbuilt functions for arrays that are usefull to manipulate pictures. Arrays used for `NumPy` can be converted to and from `OpenCV` . Therefore, any operation on arrays can be converted into `OpenCV` images.
Let's try some lines and create some arrays:

```
>>> import numpy as np
>>> x=np.array([1,2,3])
```

[1]The Picamera support has been basically dropped to shift toward a fully open source library. Problem is, the new one does not support Python's API yet. It's all for the better, but it seriously broke compatibility in the meantime.
[2]`https://github.com/PacktPublishing/raspberry-pi-computer-vision-programming`

```
>>> x
array([1, 2, 3])

>>> y=np.arange(10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

`Linspace(a,b,c)` creates an array of `c` equally spaced elements starting from `a` and ending by `b`.

```
>>> a=np.array([1,3,6,9])
>>> b=np.linspace(0,15,4)
>>> b
array([ 0., 5., 10., 15.])
>>> c=a-b
>>> c
array([ 1., -2., -4., -6.])
```

Following is the code that can be used to calculate the square of every element in an array:

```
>>> a**2
array([ 1, 9, 36, 81])
```

Many others functions and packages are available in `NumPy` such as linear algebra functions. For more information, you can visit `https://numpy.org/doc/stable/reference/index.html#reference`.

Step 2 – Image importation and exportation with `OpenCV`

`OpenCV` stands for Open Source Computer Vision and is a library that handles images in a broad way. Let's get started with the basics of `OpenCV` 's Python API. All the scripts we will write and run will be done using the `OpenCV` library, which must be imported with the line `import cv2`.
To start our manipulations, we will need the help of parrots.

You can download it at `https://raw.githubusercontent.com/parastuffs/4EISA-image-processing/main/Labo%201/parrots.bmp`, but really any image will do at this stage.
The `cv2.imread(file, mode)` method is used to import an image. It takes two arguments. The first argument `file` is the image filename. The image should either be in the same directory where the Python script is, or the absolute path should be provided to `cv2.imread()`. It reads images and saves it as a `NumPy` array. The second argument is a flag which specifies the mode the image should be read in. The flag can have following values:

`cv2.IMREAD_COLOR` or 1: This loads a colour image. This is the default flag.

`cv2.IMREAD_GRAYSCALE` or 0: This loads an image in greyscale mode.

`cv2.IMREAD_UNCHANGED` or -1: This loads an image as it is, including the alpha channel.

Write following code in a new python script:

```python
#!/usr/bin/env python3
#This imports opencv
import cv2
#This reads and stores image in color into variable img
img = cv2.imread('parrots.bmp',cv2.IMREAD_COLOR)
img = cv2.imread('parrots.bmp',1)
```

The last two lines are doing exactly the same thing.
The following code is used to display an image[3]:

```python
cv2.imshow('Parrots',img)
cv2.waitKey(0)
cv2.destroyWindow('Parrots')
```

The `cv2.imshow(window, image)` function is used to display an image. The first argument is a string, which is the window name, and the second argument is the variable that holds the image which is to be displayed.

The `cv2.waitKey(time)` function is a keyboard function. Its argument is the time in milliseconds. The function waits for specified milliseconds for any keyboard key press. If 0 is passed, it waits indefinitely for a key press. It is the only method to fetch and handle events. We must use this after `cv2.imshow()`, otherwise we don't have enough time to see the image.

The `cv2.destroyWindow()` function takes a window name as a parameter and destroys that window. If we want to destroy all the windows in the current program, we can use `cv2.destoyAllWindows()`.

Now, you know how to import and display an image, it can be useful to save it. The `cv2.imwrite(file, image)` method is used to save an image to a specific path. The first argument is the name of the file and the second is the variable pointing to the image we want to save.

[3]Don't forget the shebang and to import `cv2`

```python
import cv2
img = cv2.imread('parrots.bmp',1)
cv2.imshow('Parrots',img)
cv2.waitKey(0)
cv2.imwrite('output.jpg',img)
cv2.destroyWindow('Parrots')
```

NumPy was introduced at the step 1 as a powerful tool to manipulate arrays. To start, you can execute the following code:

```python
import cv2
import matplotlib.pyplot as plt
# Program to load a color image in gray scale and to display it using matplotlib
img = cv2.imread('parrots.bmp',0)
plt.imshow(img,cmap='gray')
plt.title('Parrots')
plt.xticks([])
plt.yticks([])
plt.show()
```

> Try again the code but without the argument `cmap='gray'`. What happens to the parrots?

The colours are all wrong because the `cv2.imread()` function of `OpenCV` reads images and saves it as a `NumPy` array of Blue, Green, and Red (BGR) pixels.

However, `plt.imshow()` displays images in RGB format. So, if we read the image as it is with `cv2.imread()` and display it using `plt.imshow()`, the value of the Blue colour will be treated as the value of Red and vice versa by `plt.imshow()`, and it would display the image with distorted colours.

To remedy this issue, we need to convert the image read in the BGR format into an RGB array format by `cv2.imread()` so that `plt.imshow()` will be able to render it in a way that makes sense to us. We will use the `cv2.cvtColor()` function for this in the step 4 and not worry about it for the moment.

`OpenCV` also has drawing functions to create dots, lines, rectangles, circles, . . . It can be combined with NumPy to display and operate on the shapes. To do so, import the necessary libraries with the following lines:

```python
import cv2
import numpy as np
```

Set a background image, for instance 400x400 black background as (0,0,0) is the black colour, with this line:

```
image = np.zeros((400,400,3), np.uint8)
```

Here is some other examples of shapes:

```
# Blue (255,0,0) rectangle with the bottom left corner at (20,20)
# and the top right at (60,60).
cv2.rectangle(image,(20,20),(60,60),(255,0,0),1)

# Green circle centered on (80,80) with a radius of 10.
cv2.circle(image,(80,80),10,(0,255,0),-1)

# Ellipse without any rotation, centered on (99,99)
# and major and minor axis length of 40 and 20.
cv2.ellipse(image,(99,99),(40,20),0,0,360,(128,128,128),-1)

# Generic polygon with four points.
points = np.array([[100,5],[125,30],[175,20],[185,10]], np.int32)
points = points.reshape((-1,1,2))
cv2.polylines(image,[points],True,(255,255,0))
```

If you pass False as the third argument in the `polylines()` function, it would join all the points and would not print a closed shape.

We can also print text in the image with `cv2.putText()`. The following code adds text to the image with (80,180) as the bottom-left corner of the text and `HERSHEY_DUPLEX2`[4] as the font with the size of 1 and colour pink:

```
cv2.putText(image,'Test',(80,180),cv2.FONT_HERSHEY_DUPLEX,1,(255,0,255))
```

Take your time to get used to the environment and do your own draft.

Step 3 – Using Picamera in Python

Make sure the camera module is enabled, navigate through `Raspberry Menu` → `Preferences` → `Raspberry Pi Configuration` → `interfaces` → `camera enable` and reboot the board if you needed to enable it.
The following program quickly demonstrates the basic usage of the picamera module to capture a picture:

```
import picamera
import time
with picamera.PiCamera() as cam:
        cam.resolution=(1024,768)
```

[4]Explore the `OpenCV` documentation to find the other possible fonts.

```python
        cam.start_preview()
        time.sleep(5)
        cam.capture('test.jpg')
```

We have to import time and picamera modules first. The `cam.start_preview()` method starts the preview and `time.sleep(5)` waits for 5 seconds before `cam.capture()` captures and saves the image in the specified file.
There is an inbuilt function in picamera for time-lapse photography. Let's see its usage using the following program:

```python
import picamera
import time
with picamera.PiCamera() as cam:
        cam.resolution=(640,480)
        cam.start_preview()
        time.sleep(3)
        for count, imagefile in enumerate(cam.capture_continuous('image{counter:02d}.jpg')):
                print(f"Capturing and saving {imagefile}")
                time.sleep(1)
                if count == 10:
                        break
```

In the preceding code, `cam.capture_continuous()` is used to capture the time-lapse sequence using the Pi camera module.

Now, we know how to capture an image. Let's try to capture it as an array and display it with OpenCV through `cv2.imshow()`. Here are the lines to realise the process.

```python
import picamera
import picamera.array
import time
import cv2
with picamera.PiCamera() as camera:
        rawCap=picamera.array.PiRGBArray(camera)
        camera.start_preview()
        time.sleep(6)
        camera.capture(rawCap,format="bgr")
image=rawCap.array
cv2.imshow("Test",image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Step 4 – Basic image processing

First of all, to process an image it is interesting to know the properties of the image. to obtain them we simply need the following code:

```python
import cv2
img = cv2.imread('parrots.bmp',1)
print(img.shape)
print(img.size)
print(img.dtype)
```

The `img.shape()` function returns the shape of an image, that is, its dimensions and the number of color channels (usually 3, corresponding to RGB).

Try modifying the preceding code to read the image in greyscale mode and observe the output of `img.shape`.

The `img.size` function returns the total number of pixels and `img.dtype` returns the image data type. Surprisingly (or not), we can perform directly arithmetic operations on images with `OpenCV` as it is an array.

Images must be of the same size for you to perform arithmetic operations on the images, and these operations are performed on individual pixels.

`cv2.add(a, b)` This function is used to add two images, where the images are passed as parameters.

`cv2.subtract(a, b)` This function is used to subtract image `b` from image `a`.

Here is an example of use:

```python
import cv2
img1 = cv2.imread('parrots.bmp',1)
img2 = cv2.imread('another_img.bmp',1)
cv2.imshow('Image1',img1)
cv2.waitKey(0)
cv2.imshow('Image2',img2)
cv2.waitKey(0)
cv2.imshow('Addition',cv2.add(img1,img2))
cv2.waitKey(0)
cv2.imshow('Image1-Image2',cv2.subtract(img1,img2))
cv2.waitKey(0)
cv2.imshow('Image2-Image1',cv2.subtract(img2,img1))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Like Whiskies, you can also blend your images. The `cv2.addWeighted()` function calculates the weighted sum of two images. Because of the weight factor, it provides a blending effect to the images. Add the following lines of code before `destroyAllWindows()` in the previous code listing to see this function in action:

```python
cv2.addWeighted(img1,0.5,img2,0.5,0)
cv2.waitKey(0)
```

In the preceding code, we passed the following five arguments to the `addWeighted()` function:

`Img1` first image

`Alpha` weight factor for the first image (0.5 in the example)

`Img2` second image

`Beta` weight factor for the second image (0.5 in the example)

`Gamma` scalar offset value (0 in the example)

The output image value is calculated with the following formula:

$$\text{Output} = (\alpha * \texttt{img1}) + (\beta * \texttt{img2}) + \gamma$$

This operation is performed on every individual pixel.
We can create a film-style transition effect on the two images by using the same function. Check out the output of the following code that creates a smooth image transition from an image to another image:

```python
import cv2
import numpy as np
import time
img1 = cv2.imread('parrots.bmp',1)
img2 = cv2.imread('another_img.bmp',1)
for i in np.linspace(0,1,40):
        alpha = i
        beta = 1-alpha
        print(f"ALPHA = {alpha}, BETA = {beta}")
        cv2.imshow('Image Transition',
        cv2.addWeighted(img1,alpha,img2,beta,0))
        time.sleep(0.05)
        if cv2.waitKey(1) == 27 :
                break
cv2.destroyAllWindows()
```

On several occasions, we may be interested in working separately with the red, green, and blue channels. For example, we might want to build a histogram for every channel of an image. Here, `cv2.split()` is used to split an image into three different intensity arrays for each colour channel, whereas `cv2.merge()` is used to merge different arrays into a single multi- channel array, that is, a colour image. The following example demonstrates this:

```python
import cv2
img = cv2.imread('parrots.bmp',1)
b,g,r = cv2.split(img)
cv2.imshow('Blue Channel', b)
cv2.imshow('Green Channel', g)
```

```
cv2.imshow('Red Channel', r)
img = cv2.merge((b,g,r))
cv2.imshow('Merged Output', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The preceding program first splits the image into three channels (blue, green, and red) and then displays each one of them. The separate channels will only hold the intensity values of the particular colour and the images will essentially be displayed as greyscale intensity images. Then, the program merges all the channels back into an image and displays it.

Creating a negative of an image

In mathematical terms, the negative of an image is the inversion of colours. For a greyscale image, it is even simpler! The negative of a greyscale image is just the intensity inversion, which can be achieved by finding the complement of the intensity from 255. A pixel value ranges from 0 to 255, and therefore, negation involves the subtracting of the pixel value from the maximum value, that is, 255. The code for the same is as follows:

```
import cv2
img = cv2.imread('parrots.bmp')
grayscale = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
negative = abs(255-grayscale)
cv2.imshow('Original',img)
cv2.imshow('Grayscale',grayscale)
cv2.imshow('Negative',negative)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Logical operations on images

OpenCV provides bitwise logical operation functions for images. We will have a look at the functions that provide the bitwise logical AND, OR, XOR (exclusive OR), and NOT (inversion) functionality. These functions can be better demonstrated visually with greyscale images. We will use barcode images in horizontal and vertical orientation for demonstration.

Let's have a look at the following code:

```
import cv2
import matplotlib.pyplot as plt
img1 = cv2.imread('Barcode_Hor.png',0)
img2 = cv2.imread('Barcode_Ver.png',0)
not_out=cv2.bitwise_not(img1)
and_out=cv2.bitwise_and(img1,img2)
```

```python
or_out=cv2.bitwise_or(img1,img2)
xor_out=cv2.bitwise_xor(img1,img2)
titles = ['Image 1','Image 2','Image 1 NOT','AND','OR','XOR']
images = [img1,img2,not_out,and_out,or_out,xor_out]
for i in xrange(6):
        plt.subplot(2,3,i+1)
        plt.imshow(images[i],cmap='gray')
        plt.title(titles[i])
        plt.xticks([]),plt.yticks([])
plt.show()
```

We first read the images in greyscale mode and calculated the NOT, AND, OR, and XOR, functionalities and then with `matplotlib`, we displayed those in a neat way. We leveraged the `mplt.subplot()` function to display multiple images. Here in the preceding example, we created a grid with two rows and three columns for our images and displayed each image in every part of the grid. You can modify this line and change it to `mplt.subplot(3,2,i+1)` to create a grid with three rows and two columns. We will use this technique heavily throughout the book to display images side-by-side or in a grid.

You may want to have a look at the functionality of `cv2.copyMakeBorder()`. This function is used to create the borders and paddings for images, and many of you will find it useful for your projects.

Challenge

For this part you are free to experience any program with `OpenCV` and the PiCamera. Navigate on the `OpenCV` API: `https://docs.opencv.org/4.x/` and the PiCamera documentation: `https://picamera.readthedocs.io/`.

As a good start try to take a picture of the EuroRobot Playground and isolate the different area of the player 1 and player 2. Then find a way to isolate the different pathways. Your challenge is to analyse the playground in depth and to obtain as much information as possible from it.