NED University of Engineering and Technology

Department of Software Engineering

Computer Vision Fundamentals (SE-488)

Lab Manual

Name: ____________________________________

Roll Number: _______________________________

Batch : ____________________________________

Year : _____________________________________

Table of Contents

Lab 1

setup OpenCV-Python in your Windows system.

Opencv Introduction

What is OpenCV? Those in the field of Artificial Intelligence (AI) and Machine Learning (ML) certainly have to learn more about it. In this blog, we shall look at what OpenCV is, how it works and where it is used today. And before we delve further into what OpenCV means, let us look at the meaning of the term computer visions and image processing.

What is Computer Vision?

Computer vision refers to a process which enables one to understand images and videos, how they are stored and how they can be manipulated and data retrieved from them. More often than not, computer vision forms the basis for AI. Today, it has an important role to play in self-driving cars, robotics and even in photo correction apps.

What is Image Processing?

Image processing refers to the analysis and manipulation of a digitized image, especially in order to improve its quality. In other words, it involves operations performed on images so as to get their enhanced versions or to glean/ extract some useful information from them. Typically such operations comprise the steps of importing the image, analyzing and manipulating it and getting the output that is typically a modified image or a report based on the image analysis.

As a result, image processing helps with tasks such as reading and writing images, detection of faces and its features, detection of shapes such as circles, squares, rectangles in an image (for instance, detection of coins in images), text recognition in images (reading vehicle number plates), changing image quality and colours / filters (think apps like Instagram) and developing Augmented Reality functionality or apps.

This brings us now to OpenCV, the focus of this blog post. So what is OpenCV?

What is OpenCV?

OpenCV stands for Open Source Computer Vision. To put it simply, it is a library used for image processing. In fact, it is a huge open-source library used for computer vision applications, in areas powered by Artificial Intelligence or Machine Learning algorithms, and for completing tasks that need image processing. As a result, it assumes significance today in real-time operations in today’s systems. Using OpenCV, one can process images and videos to identify objects, faces, or even the handwriting of a human.

Originally developed by Intel, OpenCV was later supported by Willow Garage then Itseez, in turn later acquired by Intel. The first OpenCV version was 1.0. Released under a BSD license, this cross-platform library is free for both academic and commercial use under the open-source Apache 2 License. It has C++, C, Python, Java and MATLAB (a proprietary multi-paradigm programming language that offers a good numeric computing environment) interfaces and the API for these interfaces can be found in the online documentation. OpenCV also supports Windows, Linux, Mac OS, iOS and Android. Initially, the main aim of creating OpenCV was real-time applications for computational efficiency. Since 2011, OpenCV also offers GPU acceleration for real-time operations. Upon integration with other libraries, such as NumPy, Python can process the OpenCV array structure for analysis. Identifying image patterns and its several features needs use of vector space and carrying out mathematical operations on these features.

Applications of OpenCV

There are lots of applications which rely on the use of the Open Source Computer Vision library. Let's look at them.

Face detection / face recognition/ facial recognition system

A facial recognition system is a technology that can match a human face from a digital image or a video frame against a database of faces. It is used to authenticate users through ID verification services, and works by pinpointing and measuring facial features from a given image.

Egomotion estimation

Egomotion refers to the 3D motion of a camera within an environment. In the context of computer vision, egomotion deals with estimating a camera's motion relative to a rigid scene. An example of egomotion estimation would be estimating a car's moving position with respect to lines on the road or street signs being seen from the car itself. The estimation of egomotion is important in autonomous robot navigation applications.

Gesture recognition

A subdiscipline of computer vision, gesture recognition works with the goal of interpreting human gestures via mathematical algorithms. Gestures can stem from any bodily motion or state but commonly originate from the face or hand.

Human–computer interaction (HCI)

Human-computer interaction (HCI) is a field of research in the design and the use of computer technology, which studies in detail the interfaces between people (users) and computers. HCI researchers focus on the ways in which humans interact with computers and design technologies allowing humans to interact with computers in novel ways.

Mobile robotics

A mobile robot is a robot that can move in its surroundings. Thus mobile robotics is a subfield of robotics and information engineering.

Motion understanding

As the term suggests, motion understanding refers to understanding, interpreting and overall analysing the motion of objects in certain environments, such that we are able to better predict their movements and and interact better with them.

Object detection

Object detection is a computer technology related to computer vision and image processing that focuses on identifying instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos.

Image segmentation and recognition

In digital image processing and computer vision, image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as image objects). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze.

Stereopsis stereo vision

Here stereopsis refers to the perception of depth and 3-dimensional structure formed on the basis of visual information gained from two cameras.

Structure from motion (SFM)
Structure from motion (SFM) is a photogrammetric range imaging technique for estimating three-dimensional structures from two-dimensional image sequences that can be combined with local motion signals. It is studied in the fields of computer vision and visual perception.

Motion tracking
Also known as video tracking, motion tracking is the process of locating a moving object (or multiple objects) over time using a camera.

Augmented Reality (AR)
Augmented Reality (AR) is basically an interactive experience of a real-world environment where the objects in the real world get enhanced by computer-generated perceptual information, often across several sensory faculties, be it visual, auditory, haptic, somatosensory and olfactory.

Ideal Use Cases for OpenCV library
Thanks to the above applications, there are many ideal use cases for OpenCV including (and not limited to):

1. Automated inspection and surveillance

2. Tracking the number of people in a place (for instance, the foot traffic in a mall)

3. Vehicle count on highways along with their speeds

4. Street view image stitching

5. Defect detection in manufacturing processes

6. Interactive art installations

7. Video/image search and retrieval

8. Object recognition

9. TV Channels advertisement recognition

10. Robot and driver-less car navigation and control

11. Analyzing medical images

12. Figuring out 3D structure from motion captured in videos/ films

Advantages of using OpenCV
Ease of use: OpenCV is easy and simple to learn

Availability of many tutorials: The fact that there are lots of tutorials available is a big plus as one can access many learning resources.

Compatibility with leading coding languages: OpenCV works with almost all the leading programming languages today, including Python, C++ and Java.

Free to use: Undoubtedly, a big plus is the fact that it is open source and hence free to use.

Installation Guide

Below steps are tested in a Windows 7-64 bit machine with Visual Studio 2010 and Visual Studio 2012. The screenshots shows VS2012.

Installing OpenCV from prebuilt binaries

1. Below Python packages are to be downloaded and installed to their default locations.
    a. Python 3.x (3.4+) or Python 2.7.x from here.
    b. Numpy package (for example, using `pip install numpy` command).
    c. Matplotlib (`pip install matplotlib`) (*Matplotlib is optional, but recommended since we use it a lot in our tutorials*).
2. Install all packages into their default locations. Python will be installed to `C:/Python27/` in case of Python 2.7.
3. After installation, open Python IDLE. Enter **import numpy** and make sure Numpy is working fine.
4. Download latest OpenCV release from GitHub or SourceForge site and double-click to extract it.
5. Goto **opencv/build/python/2.7** folder.
6. Copy **cv2.pyd** to **C:/Python27/lib/site-packages**.
7. Open Python IDLE and type following codes in Python terminal.

```
>>> import cv2 as cv
```

```
>>> print( cv.__version__ )
```

If the results are printed out without any errors, congratulations !!! You have installed OpenCV-Python successfully.

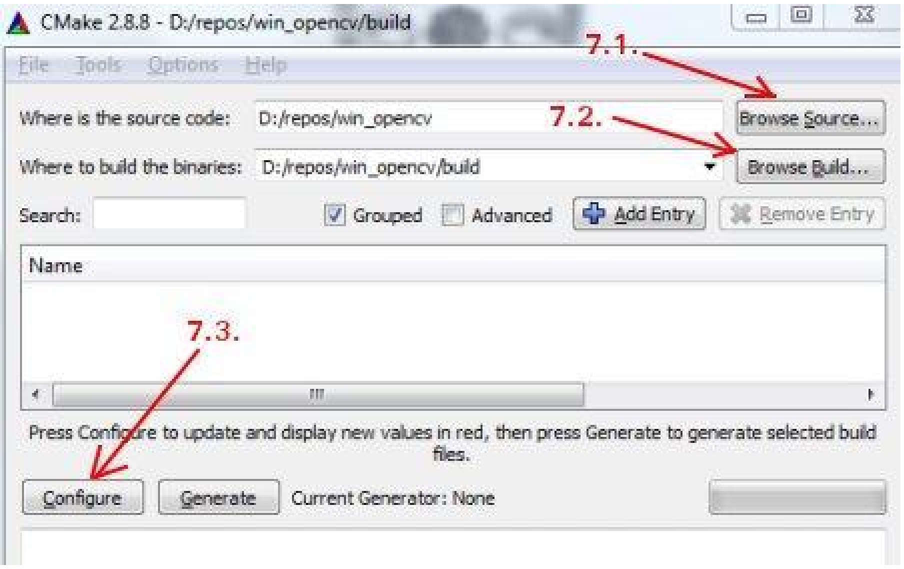
Building OpenCV from source

1. Download and install Visual Studio and CMake.
    a. Visual Studio 2012
    b. CMake
2. Download and install necessary Python packages to their default locations
    a. Python
    b. Numpy

Note

> In this case, we are using 32-bit binaries of Python packages. But if you want to use OpenCV for x64, 64-bit binaries of Python packages are to be installed. Problem is that, there is no official 64-bit binaries of Numpy. You have to build it on your own. For that, you have to use the same compiler used to build Python. When you start Python IDLE, it shows the compiler details. You can get more information here. So your system must have the same Visual Studio version and build Numpy from source.
>
> Another method to have 64-bit Python packages is to use ready-made Python distributions from third-parties like Anaconda, Enthought etc. It will be bigger in size, but will have everything you need. Everything in a single shell. You can also download 32-bit versions also.

3. Make sure Python and Numpy are working fine.
4. Download OpenCV source. It can be from Sourceforge (for official release version) or from Github (for latest source).
5. Extract it to a folder, opencv and create a new folder build in it.
6. Open CMake-gui (*Start > All Programs > CMake-gui*)
7. Fill the fields as follows (see the image below):
    a. Click on **Browse Source...** and locate the opencv folder.
    b. Click on **Browse Build...** and locate the build folder we created.
    c. Click on **Configure**.



CMake 2.8.8 - D:/repos/win_opencv/build

Where is the source code: D:/repos/win_opencv

Where to build the binaries: D:/repos/win_opencv/build

Search:

☑ Grouped □ Advanced

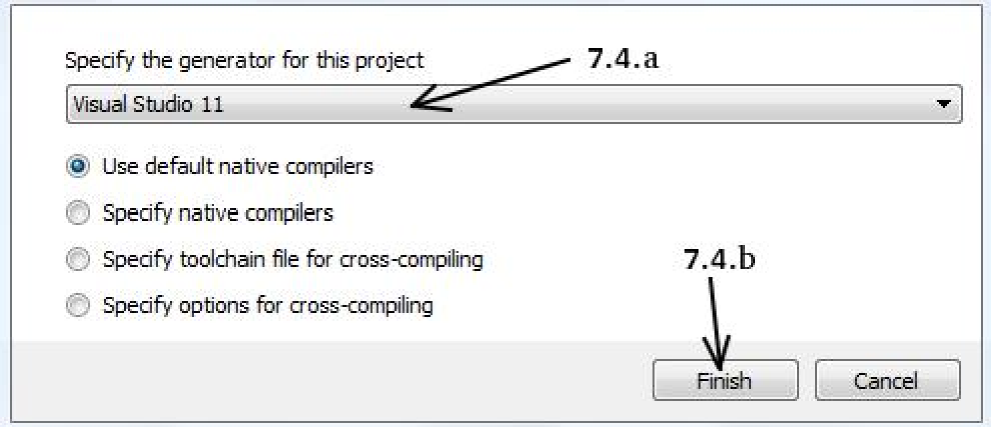
Name

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: None

image

    d. It will open a new window to select the compiler. Choose appropriate compiler (here, Visual Studio 11) and click **Finish**.



Specify the generator for this project

Visual Studio 11

● Use default native compilers

○ Specify native compilers

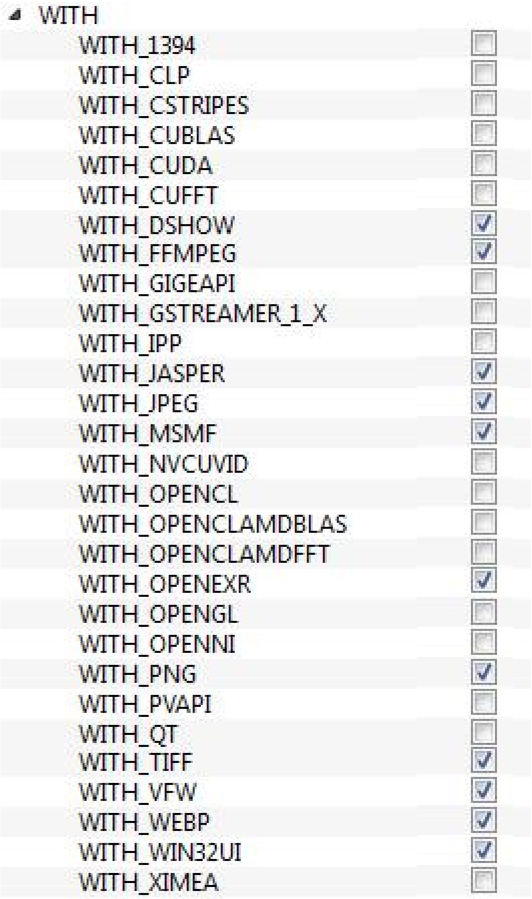
○ Specify toolchain file for cross-compiling

○ Specify options for cross-compiling

image

e. Wait until analysis is finished.

8. You will see all the fields are marked in red. Click on the **WITH** field to expand it. It decides what extra features you need. So mark appropriate fields. See the below image:

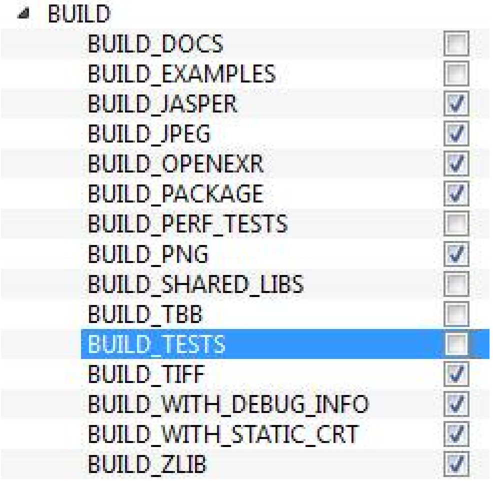


WITH

- WITH_1394 □
- WITH_CLP □
- WITH_CSTRIPES □
- WITH_CUBLAS □
- WITH_CUDA □
- WITH_CUFFT □
- WITH_DSHOW ☑
- WITH_FFMPEG ☑
- WITH_GIGEAPI □
- WITH_GSTREAMER_1_X □
- WITH_IPP □
- WITH_JASPER ☑
- WITH_JPEG ☑
- WITH_MSMF ☑
- WITH_NVCUVID □
- WITH_OPENCL □
- WITH_OPENCLAMDBLAS □
- WITH_OPENCLAMDFFT □
- WITH_OPENEXR ☑
- WITH_OPENGL □
- WITH_OPENNI □
- WITH_PNG ☑
- WITH_PVAPI □
- WITH_QT □
- WITH_TIFF ☑
- WITH_VFW ☑
- WITH_WEBP ☑
- WITH_WIN32UI ☑
- WITH_XIMEA □

image

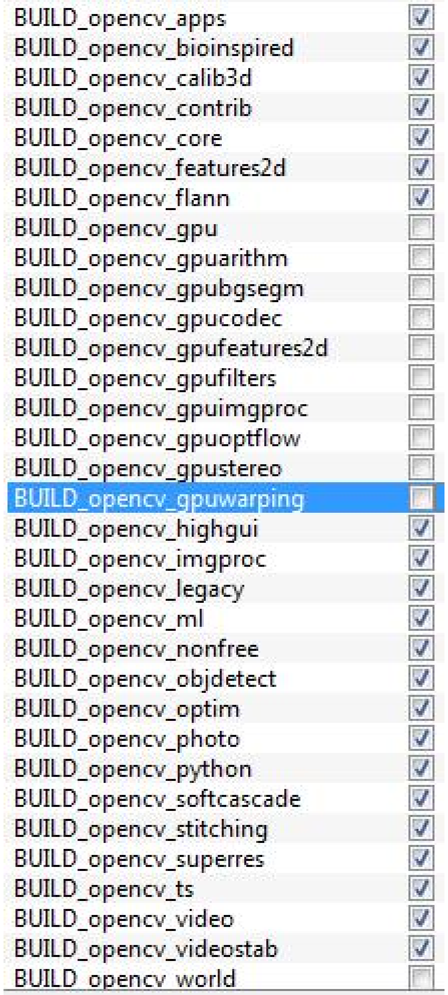
9. Now click on **BUILD** field to expand it. First few fields configure the build method. See the below image:



BUILD	
BUILD_DOCS	☐
BUILD_EXAMPLES	☐
BUILD_JASPER	☑
BUILD_JPEG	☑
BUILD_OPENEXR	☑
BUILD_PACKAGE	☑
BUILD_PERF_TESTS	☐
BUILD_PNG	☑
BUILD_SHARED_LIBS	☐
BUILD_TBB	☐
BUILD_TESTS	☐
BUILD_TIFF	☑
BUILD_WITH_DEBUG_INFO	☑
BUILD_WITH_STATIC_CRT	☑
BUILD_ZLIB	☑

image

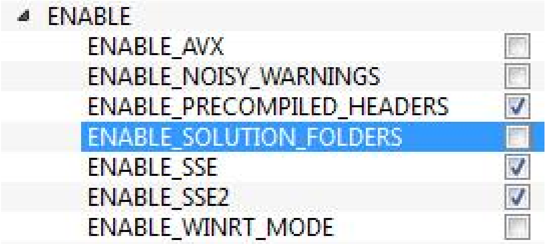
10. Remaining fields specify what modules are to be built. Since GPU modules are not yet supported by OpenCV-Python, you can completely avoid it to save time (But if you work with them, keep it there). See the image below:



BUILD_opencv_apps	☑
BUILD_opencv_bioinspired	☑
BUILD_opencv_calib3d	☑
BUILD_opencv_contrib	☑
BUILD_opencv_core	☑
BUILD_opencv_features2d	☑
BUILD_opencv_flann	☑
BUILD_opencv_gpu	□
BUILD_opencv_gpuarithm	□
BUILD_opencv_gpubgsegm	□
BUILD_opencv_gpucodec	□
BUILD_opencv_gpufeatures2d	□
BUILD_opencv_gpufilters	□
BUILD_opencv_gpuimgproc	□
BUILD_opencv_gpuoptflow	□
BUILD_opencv_gpustereo	□
BUILD_opencv_gpuwarping	□
BUILD_opencv_highgui	☑
BUILD_opencv_imgproc	☑
BUILD_opencv_legacy	☑
BUILD_opencv_ml	☑
BUILD_opencv_nonfree	☑
BUILD_opencv_objdetect	☑
BUILD_opencv_optim	☑
BUILD_opencv_photo	☑
BUILD_opencv_python	☑
BUILD_opencv_softcascade	☑
BUILD_opencv_stitching	☑
BUILD_opencv_superres	☑
BUILD_opencv_ts	☑
BUILD_opencv_video	☑
BUILD_opencv_videostab	☑
BUILD_opencv_world	□

image

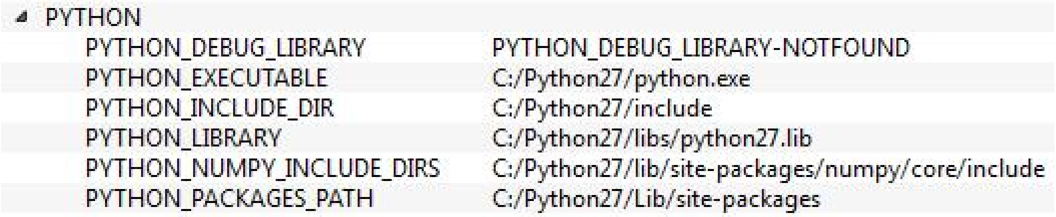
11. Now click on **ENABLE** field to expand it. Make sure **ENABLE_SOLUTION_FOLDERS** is unchecked (Solution folders are not supported by Visual Studio Express edition). See the image below:



ENABLE	
ENABLE_AVX	□
ENABLE_NOISY_WARNINGS	□
ENABLE_PRECOMPILED_HEADERS	☑
ENABLE_SOLUTION_FOLDERS	□
ENABLE_SSE	☑
ENABLE_SSE2	☑
ENABLE_WINRT_MODE	□

image

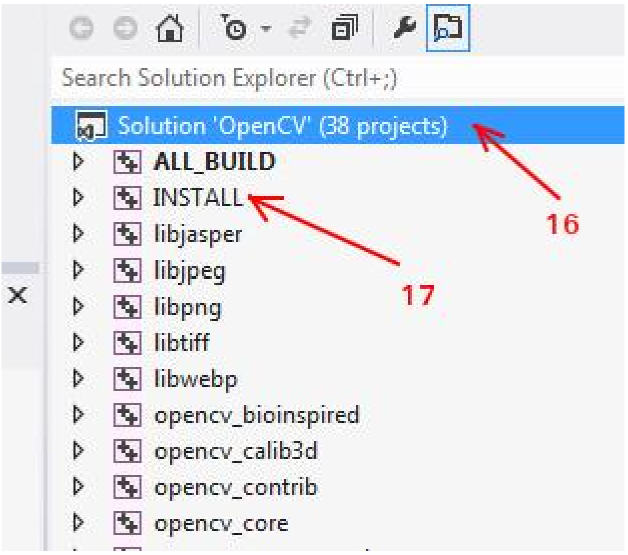
12. Also make sure that in the **PYTHON** field, everything is filled. (Ignore PYTHON_DEBUG_LIBRARY). See image below:



◢ PYTHON	
PYTHON_DEBUG_LIBRARY	PYTHON_DEBUG_LIBRARY-NOTFOUND
PYTHON_EXECUTABLE	C:/Python27/python.exe
PYTHON_INCLUDE_DIR	C:/Python27/include
PYTHON_LIBRARY	C:/Python27/libs/python27.lib
PYTHON_NUMPY_INCLUDE_DIRS	C:/Python27/lib/site-packages/numpy/core/include
PYTHON_PACKAGES_PATH	C:/Python27/Lib/site-packages

image

13. Finally click the **Generate** button.
14. Now go to our **opencv/build** folder. There you will find **OpenCV.sln** file. Open it with Visual Studio.
15. Check build mode as **Release** instead of **Debug**.
16. In the solution explorer, right-click on the **Solution** (or **ALL_BUILD**) and build it. It will take some time to finish.
17. Again, right-click on **INSTALL** and build it. Now OpenCV-Python will be installed.



image

18. Open Python IDLE and enter 'import cv2 as cv'. If no error, it is installed correctly.

Lab 2

Fundamentals of OpenCV

This lab covers opening files, looking at pixels, and some simple image processing techniques. We'll use the following sample image, stolen from the Internet. But you can use whatever image you like.

Getting Started with Images

Goals

- Here, you will learn how to read an image, how to display it and how to save it back
- You will learn these functions : cv2.imread(), cv2.imshow() , cv2.imwrite()
- Optionally, you will learn how to display images with Matplotlib

Using OpenCV

Read an image

Use the function cv2.imread() to read an image. The image should be in the working directory or a full path of image should be given.

Second argument is a flag which specifies the way 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.

See the code below:

```
import numpy as np import cv2

# Load an color image in grayscale img =
cv2.imread('messi5.jpg',0)
```

Warning: Even if the image path is wrong, it won't throw any error, but print img will give you None

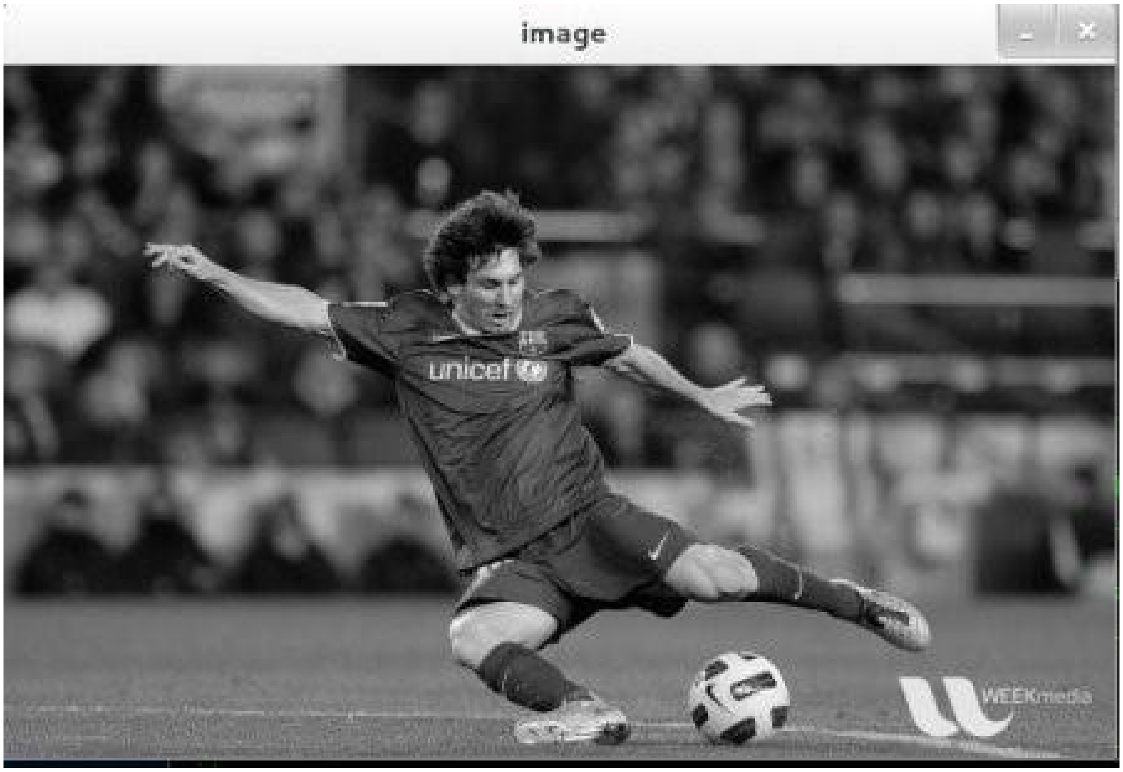
Display an image

Use the function cv2.imshow() to display an image in a window. The window automatically fits to the image size.

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()
```

A screenshot of the window will look like this (in Fedora-Gnome machine):



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 key stroke. It can also be set to detect specific key strokes like, if key *a* is pressed etc which we will discuss below. 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.

Note: There is a special case where you can already create a window and load image to it later. In that case, you can specify whether window is resizable or not. It is done with the function cv2.namedWindow(). By default, the flag is cv2.WINDOW_AUTOSIZE. But if you specify flag to be cv2.WINDOW_NORMAL, you can resize window. It will be helpful when image is too large in dimension and adding track bar to windows.

See the code below:

```
cv2.namedWindow('image', cv2.WINDOW_NORMAL)
cv2.imshow('image',img) cv2.waitKey(0)
cv2.destroyAllWindows()
```

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.

Sum it up

Below program loads an image in grayscale, displays it, save the image if you press ‘s’ and exit, or simply exit without saving if you press *ESC* key.

```
import numpy as np import cv2

img = cv2.imread('messi5.jpg',0)
cv2.imshow('image',img) k = cv2.waitKey(0)
if k == 27:                          # wait for ESC key to exit
        cv2.destroyAllWindows()
elif k == ord('s'): # wait for 's' key to save and exit
        cv2.imwrite('messigray.png',img) cv2.destroyAllWindows()
```

Warning: If you are using a 64-bit machine, you will have to modify k = cv2.waitKey(0) line as follows :

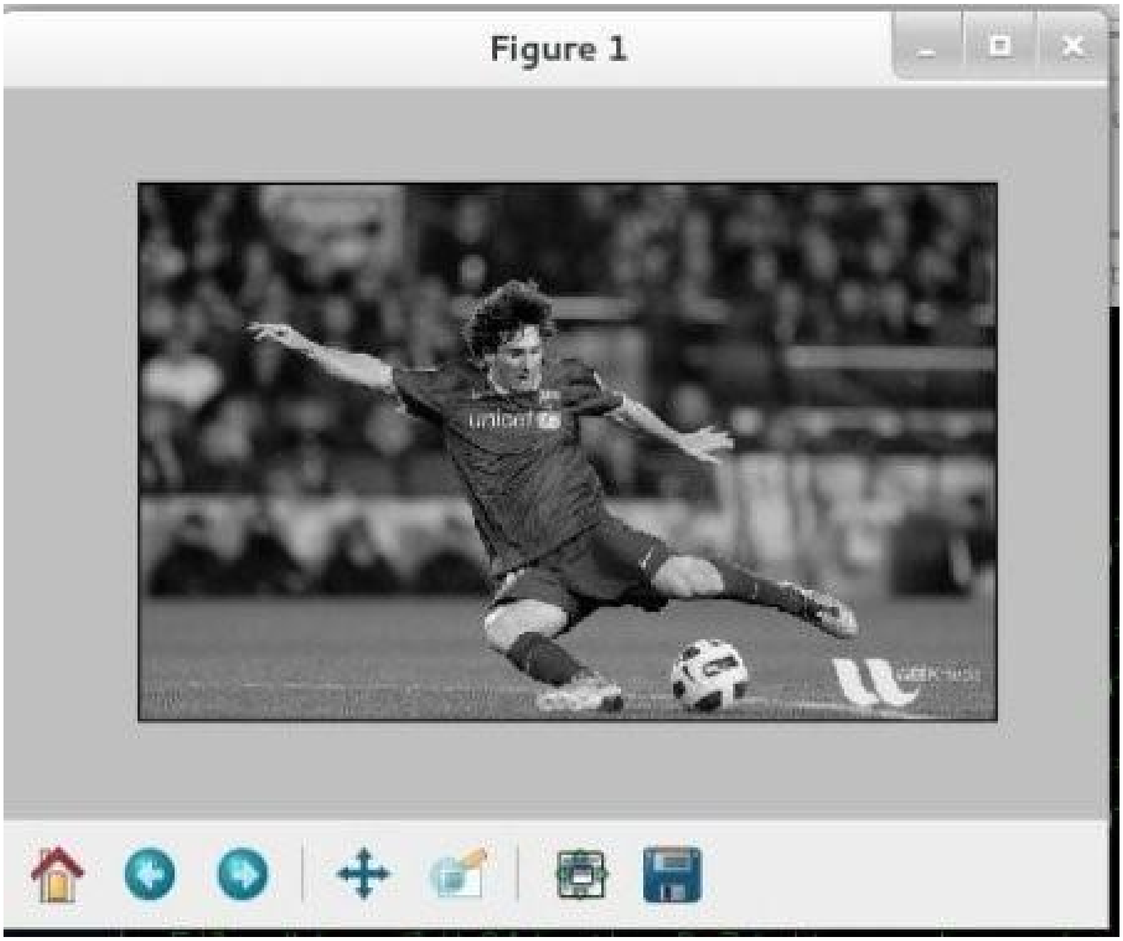
```
k = cv2.waitKey(0) & 0xFF
```

Using Matplotlib

Matplotlib is a plotting library for Python which gives you wide variety of plotting methods. You will see them in coming articles. Here, you will learn how to display image with Matplotlib. You can zoom images, save it etc using Matplotlib.

```
import numpy as np import cv2
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg',0) plt.imshow(img, cmap = 'gray', interpolation = 'bicubic') plt.xticks([]),
plt.yticks([]) # to hide tick values on X and Y axis plt.show()
```

A screen-shot of the window will look like this :



See also:

Plenty of plotting options are available in Matplotlib. Please refer to Matplotlib docs for more details. Some, we will see on the way.

Warning: Color image loaded by OpenCV is in BGR mode. But Matplotlib displays in RGB mode. So color images will not be displayed correctly in Matplotlib if image is read with OpenCV. Please see the exercises for more details.

Read Image from Urls

In this step we will read images from urls, and display them using openCV, please note the difference when reading image in RGB and BGR format. The default input color channels are in BGR format for openCV.

```
# Create a list to store the urls of the images
urls = ["https://iiif.lib.ncsu.edu/iiif/0052574/full/800,/0/default.jpg",
       "https://iiif.lib.ncsu.edu/iiif/0016007/full/800,/0/default.jpg",
      "https://placekitten.com/800/571"]
# Read and display the image
# loop over the image URLs, you could store several image urls in the list

for url in urls:
  image = io.imread(url)
  image_2 = cv.cvtColor(image, cv.COLOR_BGR2RGB)
  final_frame = cv.hconcat((image, image_2))
  cv2_imshow(final_frame)
  print('\n')
```

Getting started with python in Colab

First we need to import the relevant libraries: OpenCV itself, Numpy, and a couple of others. Common and Video are simple data handling and opening routines that you can find in the OpenCV Python Samples directory or from the github repo linked above. We'll start each notebook with the same includes - you don't need all of them every time (so this is bad form, really) but it's easier to just copy and paste.

```
# Download the test image and utils files
!wget --no-check-certificate \
https://raw.githubusercontent.com/computationalcore/introduction-to-
opencv/master/assets/noidea.jpg \
    -O noidea.jpg
!wget --no-check-certificate \
https://raw.githubusercontent.com/computationalcore/introduction-to-
opencv/master/utils/common.py \
    -O common.py
# These imports let you use opencv
import cv2 #opencv itself
import common #some useful opencv functions
import numpy as np # matrix manipulations
#the following are to do with this interactive notebook code
%matplotlib inline
from matplotlib import pyplot as plt # this lets you draw inline
pictures in the notebooks
```

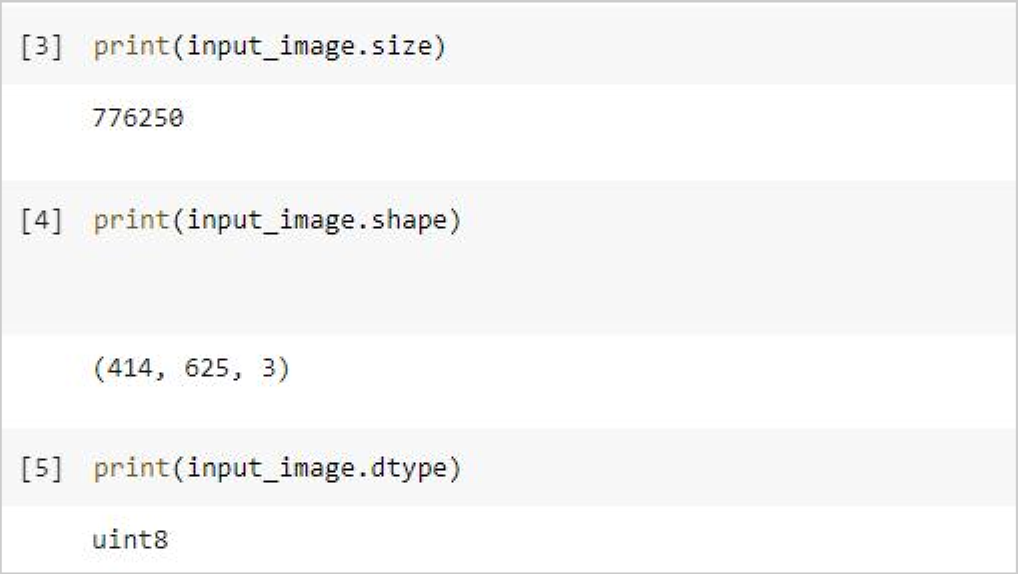
```
import pylab # this allows you to control figure size
pylab.rcParams['figure.figsize'] = (10.0, 8.0) # this controls figure size in the notebook
```

Now we can open an image:

```
input_image=cv2.imread('noidea.jpg')
```

We can find out various things about that image

```
print(input_image.size)
print(input_image.shape)
print(input_image.dtype)
```



```
[3]  print(input_image.size)

     776250

[4]  print(input_image.shape)

     (414, 625, 3)

[5]  print(input_image.dtype)

     uint8
```

‘Gotcha’ that last one (datatype) is one of the tricky things about working in Python. As it's not strongly typed, Python will allow you to have arrays of different types but the same size, and some functions will return arrays of types that you probably don't want. Being able to check and inspect the data type like this is very useful and is one of the things I often find myself doing in debugging.

```
plt.imshow(input_image)
# If you want to see them all, rather than just a count uncomment the following line
#print(COLORflags)

opencv_merged=cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB)
```

```
plt.imshow(opencv_merged)
```

Exercise:

1: Load and show image using file and urls? Attach screen shot.

Lab 3

Drawing Functions in OpenCV

Goal

- Learn to draw different geometric shapes with OpenCV
- You will learn these functions : cv2.line(), cv2.circle() , cv2.rectangle(), cv2.ellipse(), cv2.putText() etc.

Code

In all the above functions, you will see some common arguments as given below:

- img : The image where you want to draw the shapes
- color : Color of the shape. for BGR, pass it as a tuple, eg: (255,0,0) for blue. For grayscale, just pass the scalar value.
- thickness : Thickness of the line or circle etc. If -1 is passed for closed figures like circles, it will fill the shape. *default thickness = 1*
- lineType : Type of line, whether 8-connected, anti-aliased line etc. *By default, it is 8-connected.* cv2.LINE_AA gives anti-aliased line which looks great for curves.

Drawing Line

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

1.2. Gui Features in OpenCV

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

```
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 ellipse in anti-clockwise direction. startAngle and endAngle denotes the starting and ending of ellipse arc measured in clockwise direction from major axis. i.e. giving values 0 and 360 gives the full ellipse. For more details, check the documentation of cv2.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 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))
```

Note: If third argument is False, you will get a polylines joining all the points, not a closed shape.

Note: cv2.polylines() can be used to draw multiple lines. Just create a list of all the lines you want to draw and pass it to the function. All lines will be drawn individually. It is more better and faster way to draw a group of lines than calling cv2.line() for each line.

Adding Text to Images:

To put texts in images, you need specify following things.

- Text data that you want to write

- Position coordinates of where you want put it (i.e. bottom-left corner where data starts).
- Font type (Check cv2.putText() docs for supported fonts)
- Font Scale (specifies the size of font)
- regular things like color, thickness, lineType etc. For better look, lineType = cv2.LINE_AA is recommended.

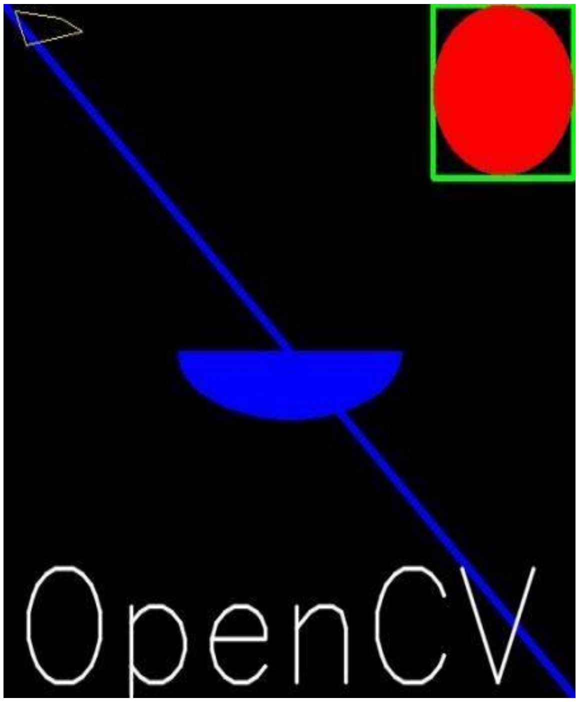
We will write OpenCV on our image in white color.

```
font = cv2.FONT_HERSHEY_SIMPLEX
cv2.putText(img,'OpenCV',(10,500), font, 4,(255,255,255),2,cv2.LINE_AA)
```

Result

So it is time to see the final result of our drawing. As you studied in previous articles, display the image to see it

1.2. Gui Features in OpenCV



Additional Resources

1. The angles used in ellipse function is not our circular angles. For more details, visit this discussion.

Exercises

1. Try to create the logo of OpenCV using drawing functions available in OpenCV

Lab 4

Basic Operations on Images

You will Learn to:

- Access pixel values and modify them
- Access image properties
- Setting Region of Image (ROI)
- Splitting and Merging images

Getting started with python

First we need to import the relevant libraries: OpenCV itself, Numpy, and a couple of others. Common and Video are simple data handling and opening routines that you can find in the OpenCV Python Samples directory or from the github repo linked above. We'll start each notebook with the same includes - you don't need all of them every time (so this is bad form, really) but it's easier to just copy and paste.

```
# Download the test image and utils files
!wget --no-check-certificate \
https://raw.githubusercontent.com/computationalcore/introduction-to-
opencv/master/assets/noidea.jpg \
    -O noidea.jpg
!wget --no-check-certificate \
https://raw.githubusercontent.com/computationalcore/introduction-to-
opencv/master/utils/common.py \
    -O common.py
# These imports let you use opencv
import cv2 #opencv itself
import common #some useful opencv functions
import numpy as np # matrix manipulations
#the following are to do with this interactive notebook code
%matplotlib inline
```

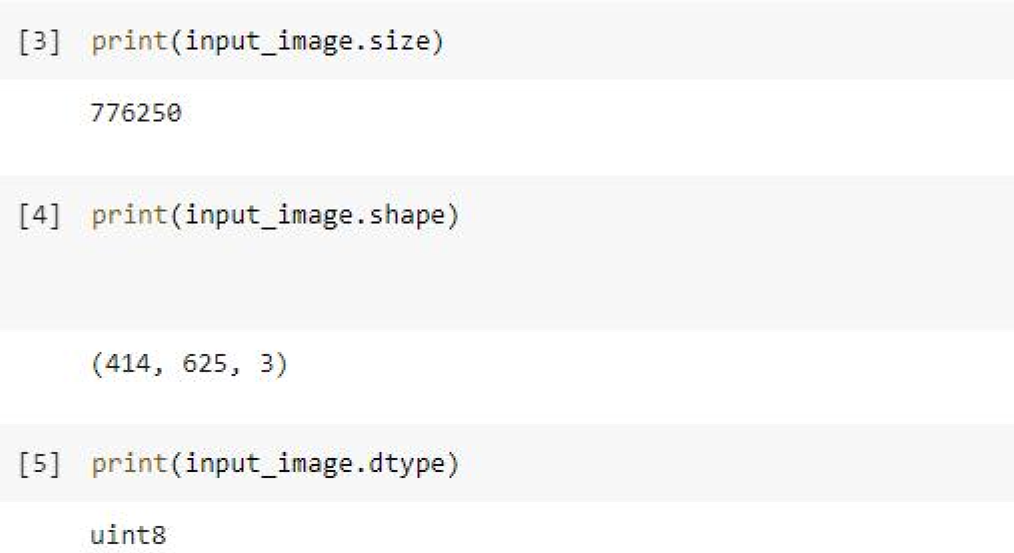
```
from matplotlib import pyplot as plt # this lets you draw inline
pictures in the notebooks
import pylab # this allows you to control figure size
pylab.rcParams['figure.figsize'] = (10.0, 8.0) # this controls
figure size in the notebook
```

Now we can open an image:

```
input_image=cv2.imread('noidea.jpg')
```

We can find out various things about that image

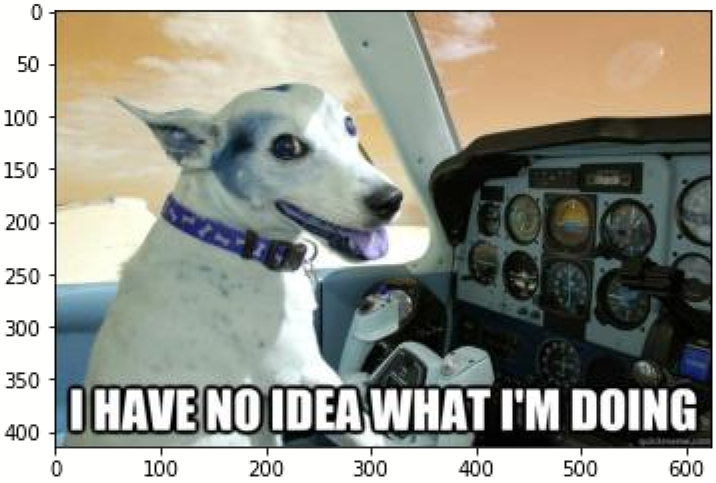
```
print(input_image.size)
print(input_image.shape)
print(input_image.dtype)
```



```
[3]  print(input_image.size)

     776250

[4]  print(input_image.shape)

     (414, 625, 3)

[5]  print(input_image.dtype)

     uint8
```

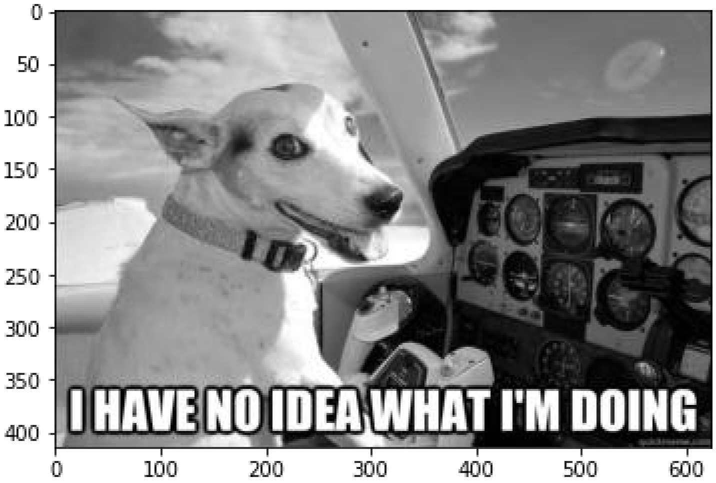
'Gotcha' that last one (datatype) is one of the tricky things about working in Python. As it's not strongly typed, Python will allow you to have arrays of different types but the same size, and some functions will return arrays of types that you probably don't want. Being able to check and inspect the data type like this is very useful and is one of the things I often find myself doing in debugging.

```
plt.imshow(input_image)
```



What this illustrates is something key about OpenCV: it doesn't store images in RGB format, but in BGR format.

```
    # split channels
    b,g,r=cv2.split(input_image)
    # show one of the channels (this is red - see that the sky is kind of dark. try changing it to b)
    plt.imshow(r, cmap='gray')
```



Converting between color spaces, merging and splitting channels

We can convert between various color spaces in OpenCV easily. We've seen how to split, above. We can also merge channels:

```
    merged=cv2.merge([r,g,b])
    # merge takes an array of single channel matrices
    plt.imshow(merged)
```

OpenCV also has a function specifically for dealing with image color spaces, so rather than split and merge channels by hand you can use this instead. It is usually marginally faster...
There are something like 250 color related flags in OpenCV for conversion and display. The ones you are most likely to use are COLOR_BGR2RGB for RGB conversion, COLOR_BGR2GRAY for conversion to grayscale, and COLOR_BGR2HSV for conversion to Hue,Saturation,Value color space.

```
COLORflags = [flag for flag in dir(cv2) if
flag.startswith('COLOR') ]
print(len(COLORflags))

# If you want to see them all, rather than just a count uncomment
the following line
#print(COLORflags)

opencv_merged=cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB)
plt.imshow(opencv_merged)
```

Getting image data and setting image data

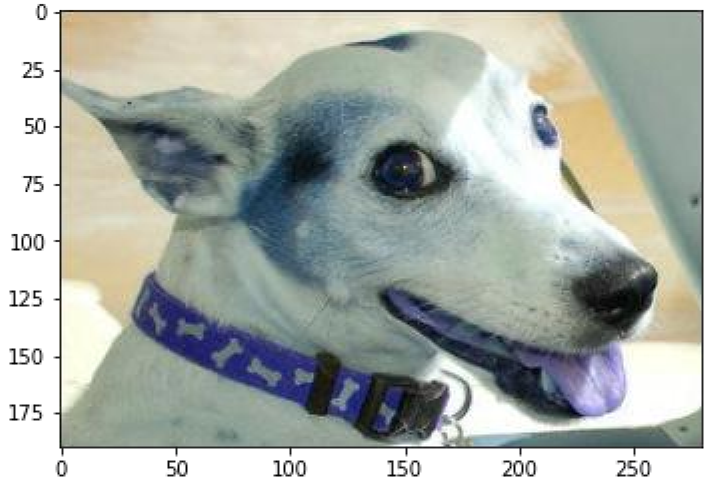
Images in python OpenCV are numpy arrays. Numpy arrays are optimized for fast array operations and so there are usually fast methods for doing array calculations which don't actually involve writing all the detail yourself. So it's usually bad practice to access individual pixels, but you can.

```
pixel = input_image[100,100]
print(pixel)

input_image[100,100] = [0,0,0]
pixelnew = input_image[100,100]
print(pixelnew)
```

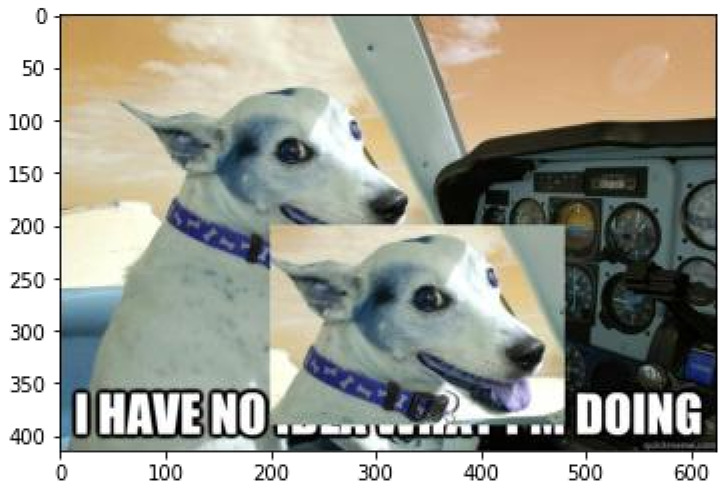
Getting and setting regions of an image

In the same way as we can get or set individual pixels, we can get or set regions of an image. This is a particularly useful way to get a region of interest to work on.

```
dogface = input_image[60:250, 70:350]
plt.imshow(dogface)
```



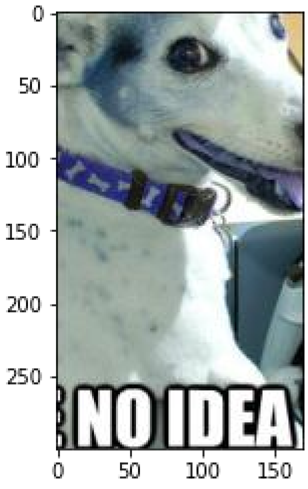
```
fresh_image=cv2.imread('noidea.jpg') # it's either start with a
fresh read of the image,
# or end up with dog faces on dogfaces on dogfaces
# as you rerun parts of the notebook but not
others...
fresh_image[200:200+dogface.shape[0],
200:200+dogface.shape[1]]=dogface
print(dogface.shape[0])
print(dogface.shape[1])
plt.imshow(fresh_image)
```



Matrix slicing

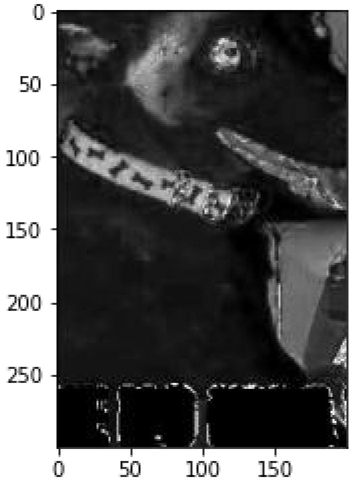
In OpenCV python style, as I have mentioned, images are numpy arrays. There are some superb array manipulation in numpy tutorials out there: this is a great introduction if you've not done it before
The getting and setting of regions above uses slicing, though, and I'd like to finish this notebook with a little more detail on what is going on there.

```
freshim2 = cv2.imread("noidea.jpg")
crop = freshim2[100:400, 130:300]
plt.imshow(crop)
```



The key thing to note here is that the slicing works like **[top_y:bottom_y, left_x:right_x]**
This can also be thought of as **[y:y+height, x:x+width]**
You can also use slicing to separate out channels. In this case you want **[y:y+height, x:x+width, channel]**
where channel represents the color you're interested in - this could be 0 = blue, 1 = green or 2=red if you're dealing with a default OpenCV image, but if you've got an image that has been converted it could be something else. Here's an example that converts to HSV then selects the S (Saturation) channel of the same crop above:

```
hsvim=cv2.cvtColor(freshim2,cv2.COLOR_BGR2HSV)
bcrop =hsvim[100:400, 100:300, 1]
plt.imshow(bcrop, cmap="gray")
```



Exercises

1. Explain the applications of python with respect to computer vision.
2. Implement a simple python program with an image of your choice and demonstrate all the concepts (attach screenshots).

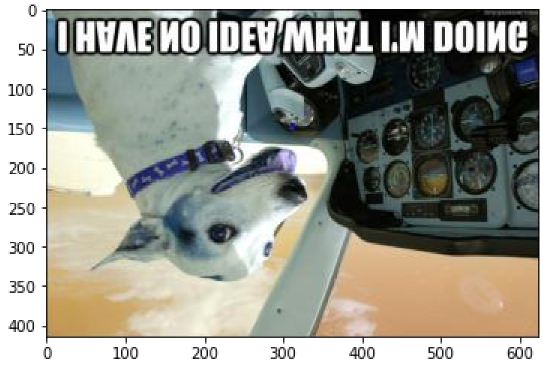
Lab 5

Image manipulations

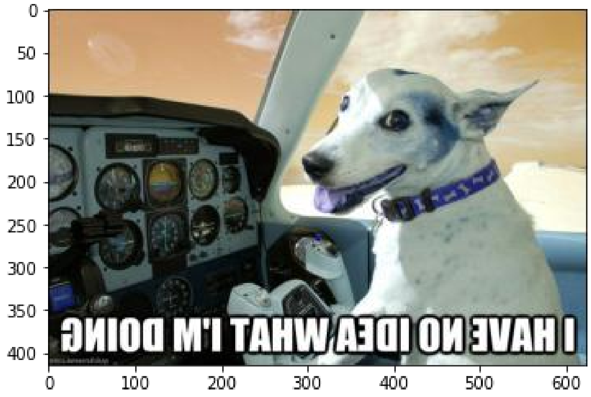
Basic manipulations

Rotate, flip...

```
flipped_code_0=cv2.flip(input_image,0) # vertical flip
plt.imshow(flipped_code_0)
```



```
flipped_code_1=cv2.flip(input_image,1) # horizontal flip
plt.imshow(flipped_code_1)
```



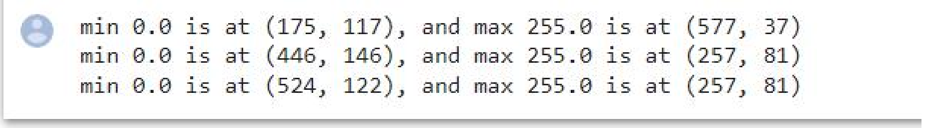
```
transposed=cv2.transpose(input_image)
plt.imshow(transposed)
```

0
100
200
300
400
500
600
0
200
400
I HAVE NO IDEA WHAT I'M DOING

Minimum, maximum

To find the min or max of a matrix, you can use minMaxLoc. This takes a single channel image (it doesn't make much sense to take the max of a 3 channel image). So in the next code snippet you see a for loop, using python style image slicing, to look at each channel of the input image separately.

```
for i in range(0,3):
   min_value, max_value, min_location,
max_location=cv2.minMaxLoc(input_image[:,:,i])
   print("min {} is at {}, and max {} is at {}".format(min_value,
min_location, max_value, max_location))
```



```
min 0.0 is at (175, 117), and max 255.0 is at (577, 37)
min 0.0 is at (446, 146), and max 255.0 is at (257, 81)
min 0.0 is at (524, 122), and max 255.0 is at (257, 81)
```

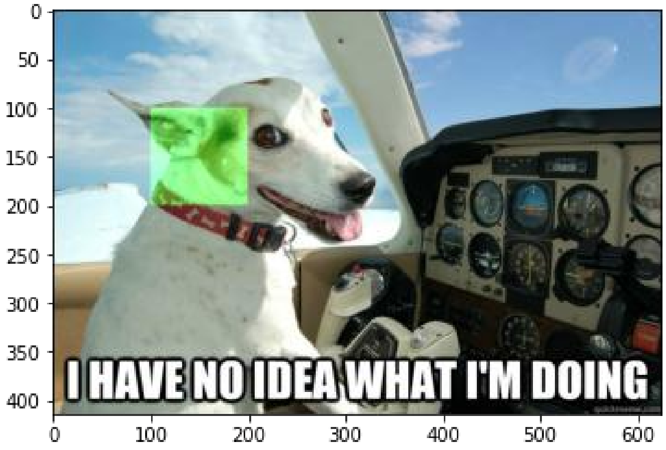
Lab 6

Arithmetic operations on images

OpenCV has a lot of functions for doing mathematics on images. Some of these have "analogous" numpy alternatives, but it is nearly always better to use the OpenCV version. The reason for this is that OpenCV is designed to work on images and so handles overflow better (OpenCV add, for example, truncates to 255 if the datatype is image-like and 8 bit; Numpy's alternative wraps around).

Useful arithmetic operations include add and addWeighted, which combine two images that are the same size.

```
#First create an image the same size as our input
blank_image = np.zeros((input_image.shape), np.uint8)
blank_image[100:200,100:200,1]=100; #give it a green square
new_image=cv2.add(blank_image,input_image) # add the two images
together
plt.imshow(cv2.cvtColor(new_image, cv2.COLOR_BGR2RGB))
```



Arithmetic Operations on Images

Goal

- Learn several arithmetic operations on images like addition, subtraction, bitwise operations etc.
- You will learn these functions : cv2.add(), cv2.addWeighted() etc.

Image Addition

You can add two images by OpenCV function, cv2.add() or simply by numpy operation, res = img1 + img2. Both images should be of same depth and type, or second image can just be a scalar value.

Note: There is a difference between OpenCV addition and Numpy addition. OpenCV addition is a saturated operation while Numpy addition is a modulo operation.

For example, consider below sample:

```
>>> x = np.uint8([250])
>>> y = np.uint8([10])

>>> print cv2.add(x,y) # 250+10 = 260 => 255
[[255]]
```

```
>>> print x+y                          # 250+10 = 260 % 256 = 4
[4]
```

It will be more visible when you add two images. OpenCV function will provide a better result. So always better stick to OpenCV functions.

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:

$$g(x) = (1 - \alpha)f_0(x) + \alpha f_1(x)$$

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

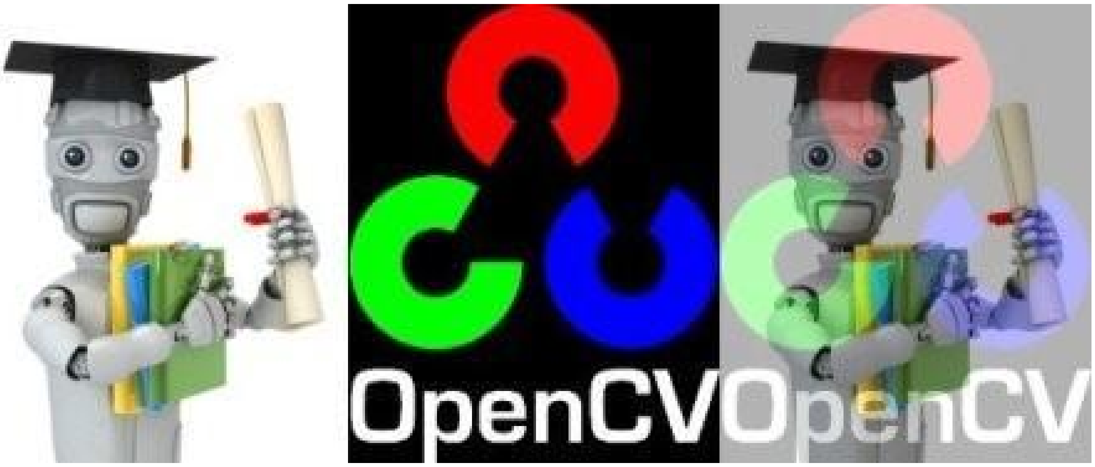
Here I took two images to blend them together. First image is given a weight of 0.7 and second image is given 0.3. cv2.addWeighted() applies following equation on the image.

$$dst = \alpha \cdot img1 + \beta \cdot img2 + \gamma$$

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()
```

Check the result below:



Exercise:

1: apply all these explained concept on you images

2: explore and apply multiplying and subtraction operation ?

3: How to apply masking using bitwise operation explain and execute on colab.

Lab 7

Smoothing Images

Goals

Learn to:

- Blur images with various low pass filters
- Apply custom-made filters to images (2D convolution)

2D Convolution (Image Filtering)

As for one-dimensional signals, images also can be filtered with various low-pass filters (LPF), high-pass filters (HPF), etc. A LPF helps in removing noise, or blurring the image. A HPF filters helps in finding edges in an image.
OpenCV provides a function, cv2.filter2D(), to convolve a kernel with an image. As an example, we will try an averaging filter on an image. A 5x5 averaging filter kernel can be defined as follows:

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Filtering with the above kernel results in the following being performed: for each pixel, a 5x5 window is centered on this pixel, all pixels falling within this window are summed up, and the result is then divided by 25. This equates to computing the average of the pixel values inside that window. This operation is performed for all the pixels in the image to produce the output filtered image. Try this code and check the result:

```
import cv2 import numpy
as np
from matplotlib import pyplot as plt

img = cv2.imread('opencv_logo.png')

kernel = np.ones((5,5),np.float32)/25 dst =
cv2.filter2D(img,-1,kernel)

plt.subplot(121),plt.imshow(img),plt.title('Original') plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(dst),plt.title('Averaging') plt.xticks([]), plt.yticks([])
plt.show()
```

Result:

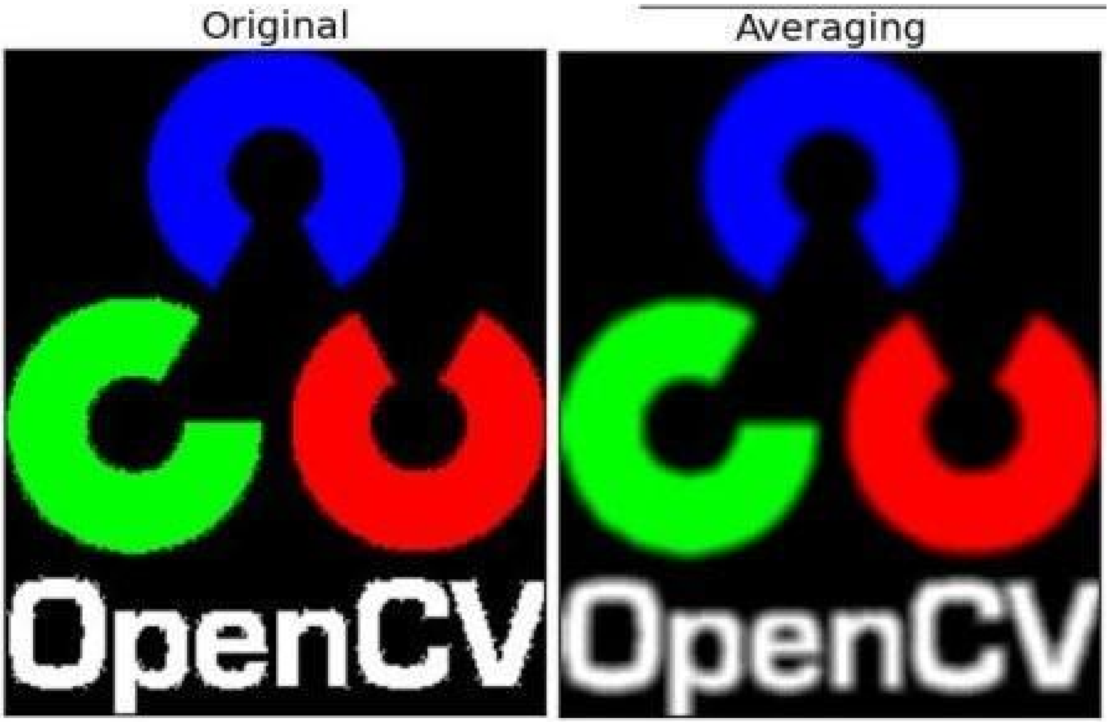


Image Blurring (Image Smoothing)

Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise. It actually removes high frequency content (e.g: noise, edges) from the image resulting in edges being blurred when this is filter is applied. (Well, there are blurring techniques which do not blur edges). OpenCV provides mainly four types of blurring techniques.

1. 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:

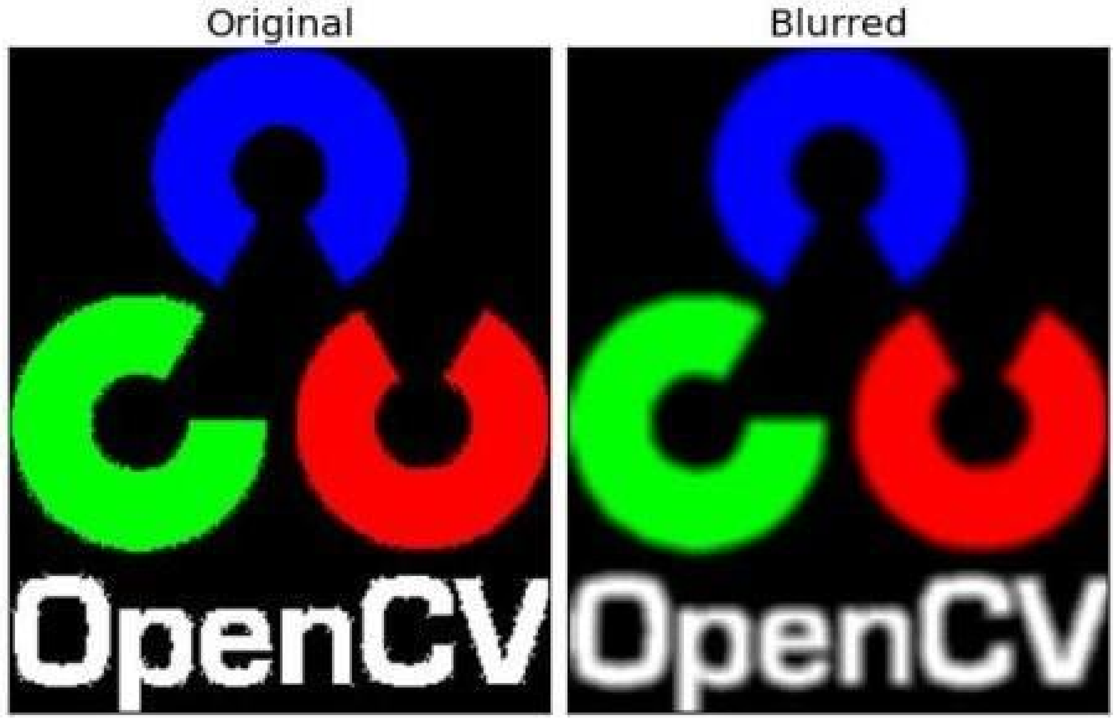
$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Note: If you don't want to use a normalized box filter, use cv2.boxFilter() and pass the argument normalize=False to the function.

Check the sample demo below with a kernel of 5x5 size:

```
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()
```

Result:



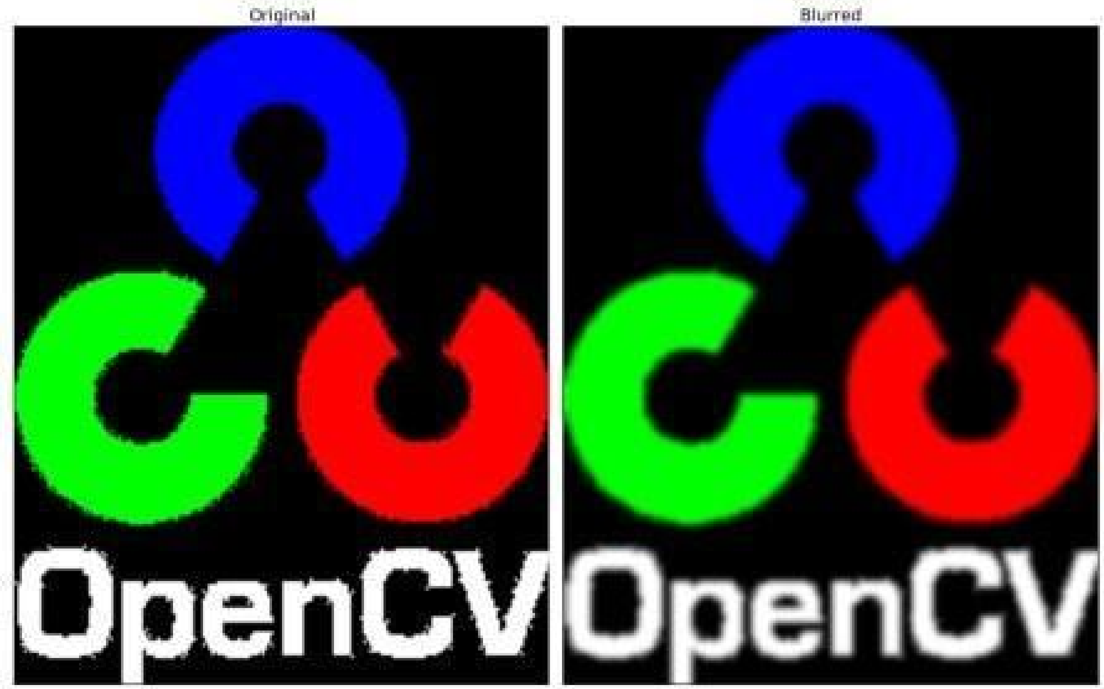
2. Gaussian Filtering

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)
```

Result:



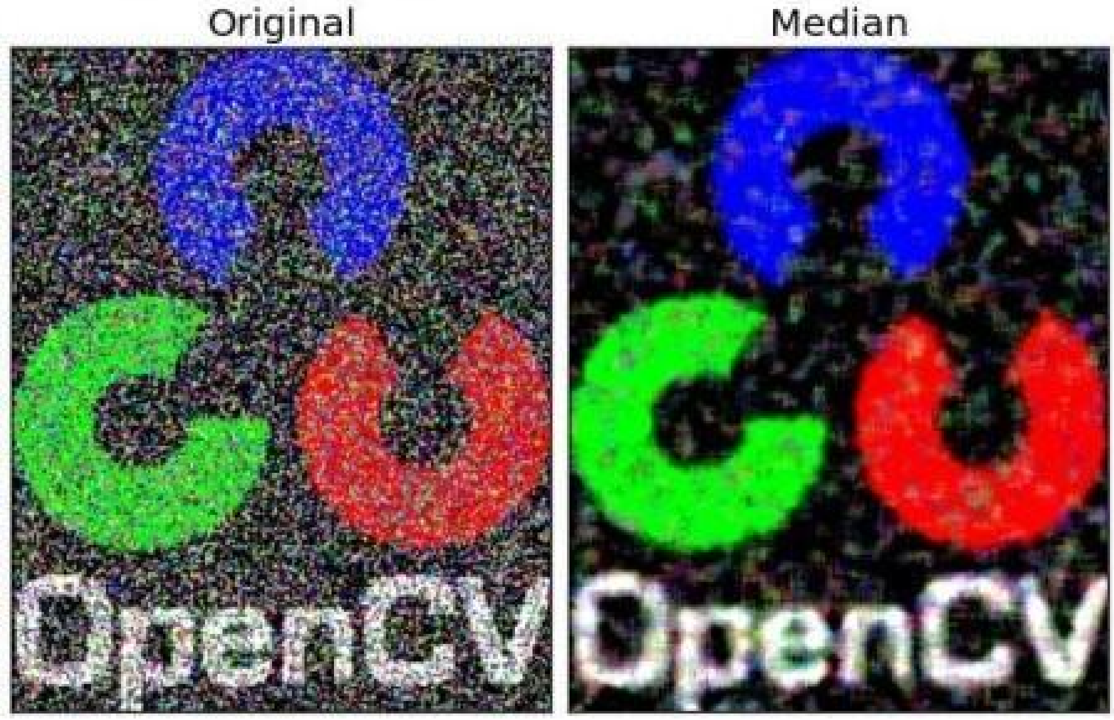
3. Median Filtering

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. Check the result:

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

Result:



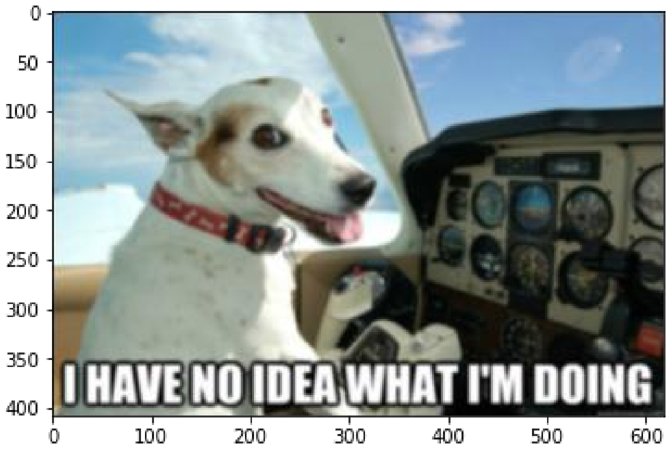
Original

Median

Colab code:

Noise reduction usually involves blurring/smoothing an image using a Gaussian kernel. The width of the kernel determines the amount of smoothing.

```
d=3
img_blur3 = cv2.GaussianBlur(input_image, (2*d+1, 2*d+1), -1)[d:-
d,d:-d]
plt.imshow(cv2.cvtColor(img_blur3, cv2.COLOR_BGR2RGB))
```



```
d=5
img_blur5 = cv2.GaussianBlur(input_image, (2*d+1, 2*d+1), -1)[d:-
d,d:-d]
```

```
plt.imshow(cv2.cvtColor(img_blur5, cv2.COLOR_BGR2RGB))
```

```
d=15
img_blur15 = cv2.GaussianBlur(input_image, (2*d+1, 2*d+1), -1)[d:-
d,d:-d]
plt.imshow(cv2.cvtColor(img_blur15, cv2.COLOR_BGR2RGB))
```

Lab 8

Image Gradients & Edge Detection

Goal

In this lab, we will learn to:

- Find Image gradients, edges etc
- We will see following functions : cv2.Sobel(), cv2.Scharr(), cv2.Laplacian() etc

Theory

OpenCV provides three types of gradient filters or High-pass filters, Sobel, Scharr and Laplacian. We will see each one of them.

1. Sobel and Scharr Derivatives

Sobel operators is a joint Gausssian smoothing plus differentiation operation, so it is more resistant to noise. You can specify the direction of derivatives to be taken, vertical or horizontal (by the arguments, yorder and xorder respectively). You can also specify the size of kernel by the argument ksize. If ksize = -1, a 3x3 Scharr filter is used which gives better results than 3x3 Sobel filter. Please see the docs for kernels used.

2. Laplacian Derivatives

It calculates the Laplacian of the image given by the relation, $\Delta src = \frac{\partial^2 src}{\partial x^2} + \frac{\partial^2 src}{\partial y^2}$ where each derivative is found using Sobel derivatives. If ksize = 1, then following kernel is used for filtering:

$$kernel = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

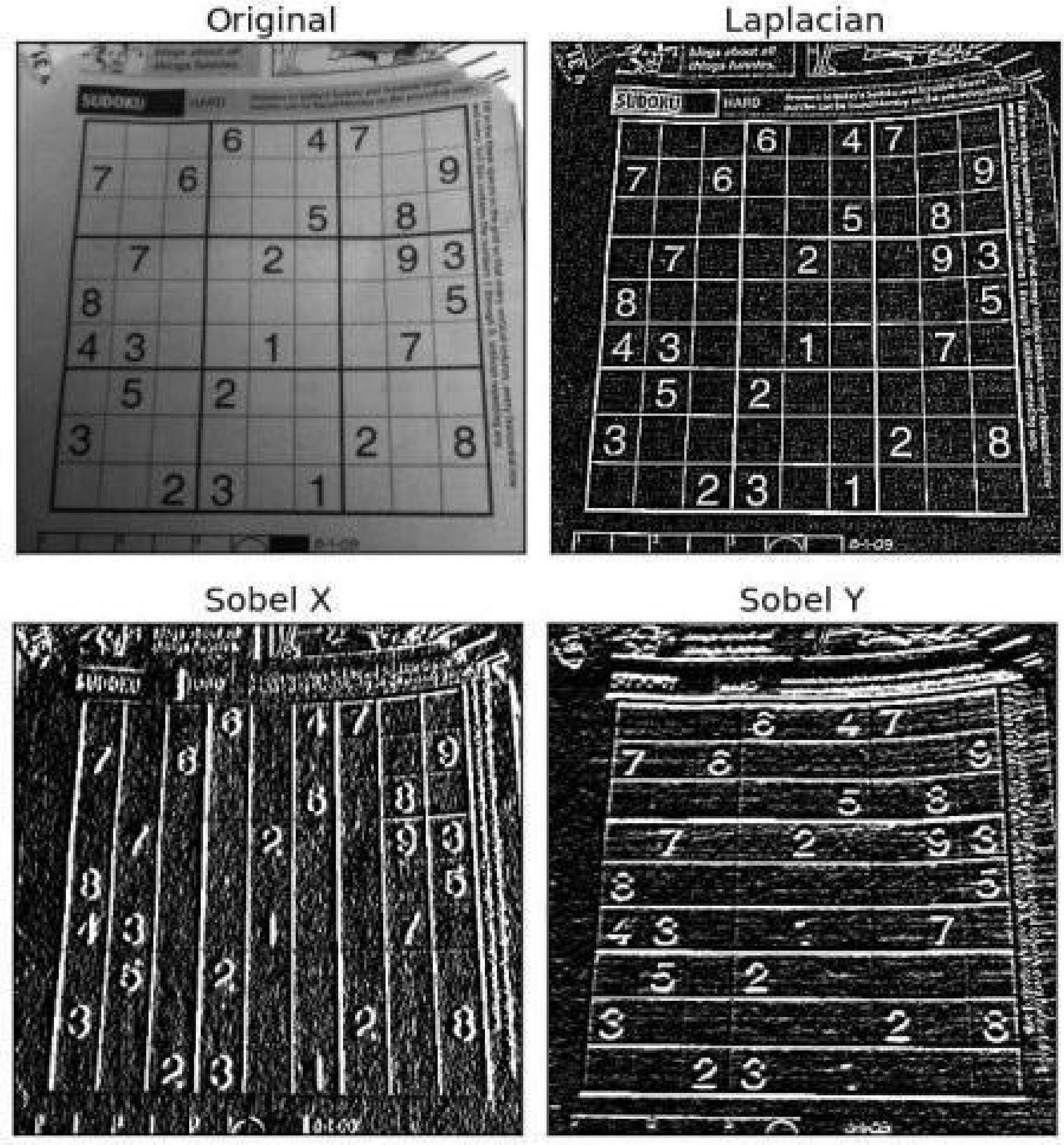
1.4. Image Processing in OpenCV

Code

Below code shows all operators in a single diagram. All kernels are of 5x5 size. Depth of output image is passed -1 to get the result in np.uint8 type.

```
import cv2 import numpy
as np
from matplotlib import pyplot as plt

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

laplacian = cv2.Laplacian(img,cv2.CV_64F) sobelx =
cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5) sobely =
cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray') plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray') plt.title('Laplacian'), plt.xticks([]),
plt.yticks([]) plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray') plt.title('Sobel X'), plt.xticks([]),
plt.yticks([]) plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray') plt.title('Sobel Y'), plt.xticks([]),
plt.yticks([])

plt.show()
```

Result:



One Important Matter!

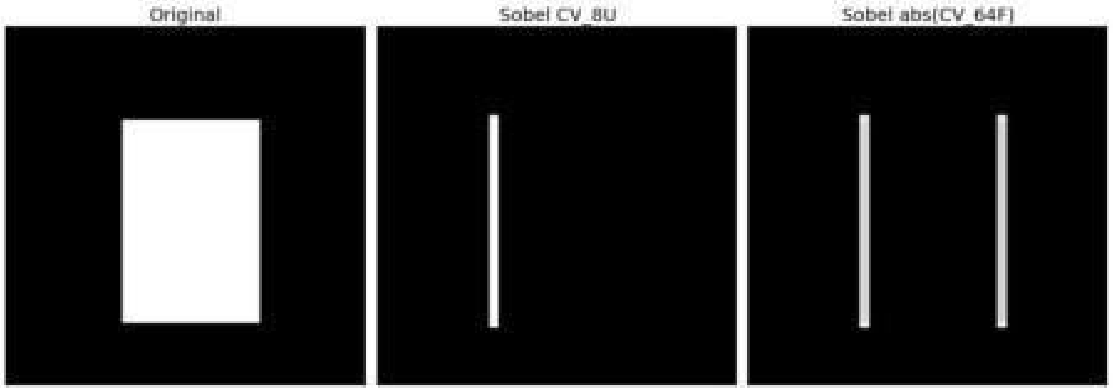
In our last example, output datatype is cv2.CV_8U or np.uint8. But there is a slight problem with that. Black-to-White transition is taken as Positive slope (it has a positive value) while White-to-Black transition is taken as a Negative slope (It has negative value). So when you convert data to np.uint8, all negative slopes are made zero. In simple words, you miss that edge.

If you want to detect both edges, better option is to keep the output datatype to some higher forms, like cv2.CV_16S, cv2.CV_64F etc, take its absolute value and then convert back to cv2.CV_8U. Below code demonstrates this procedure for a horizontal Sobel filter and difference in results.

```
import cv2 import numpy as np
from matplotlib import pyplot as
plt
```

```
img = cv2.imread('box.png',0)

# Output dtype = cv2.CV_8U sobelx8u =
cv2.Sobel(img,cv2.CV_8U,1,0,ksize=5)

# Output dtype = cv2.CV_64F. Then take its absolute and convert to cv2.CV_8U sobelx64f =
cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5) abs_sobel64f = np.absolute(sobelx64f) sobel_8u =
np.uint8(abs_sobel64f)

plt.subplot(1,3,1),plt.imshow(img,cmap = 'gray') plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(1,3,2),plt.imshow(sobelx8u,cmap = 'gray') plt.title('Sobel CV_8U'), plt.xticks([]),
plt.yticks([]) plt.subplot(1,3,3),plt.imshow(sobel_8u,cmap = 'gray') plt.title('Sobel abs(CV_64F)'),
plt.xticks([]), plt.yticks([])

plt.show()
```

Check the result below:



Original

Sobel CV_8U

Sobel abs(CV_64F)

Additional Resources

Exercises

Canny Edge Detection

Goal

In this chapter, we will learn about

- Concept of Canny edge detection
- OpenCV functions for that : cv2.Canny()

Theory

Canny Edge Detection is a popular edge detection algorithm. It was developed by John F. Canny in 1986. It is a multi-stage algorithm and we will go through each stages.

1. Noise Reduction

Since edge detection is susceptible to noise in the image, first step is to remove the noise in the image with a 5x5 Gaussian filter. We have already seen this in previous chapters.

2. Finding Intensity Gradient of the Image

Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction (G_x) and vertical direction (G_y). From these two images, we can find edge gradient and direction for each pixel as follows:

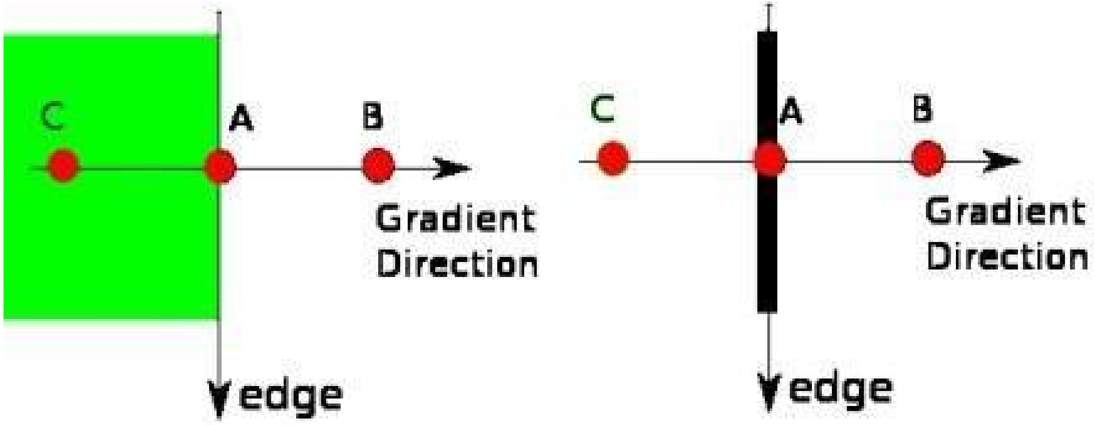
$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Gradient direction is always perpendicular to edges. It is rounded to one of four angles representing vertical, horizontal and two diagonal directions.

3. Non-maximum Suppression

After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. For this, at every pixel, pixel is checked if it is a local maximum in its neighborhood in the direction of gradient. Check the image below:

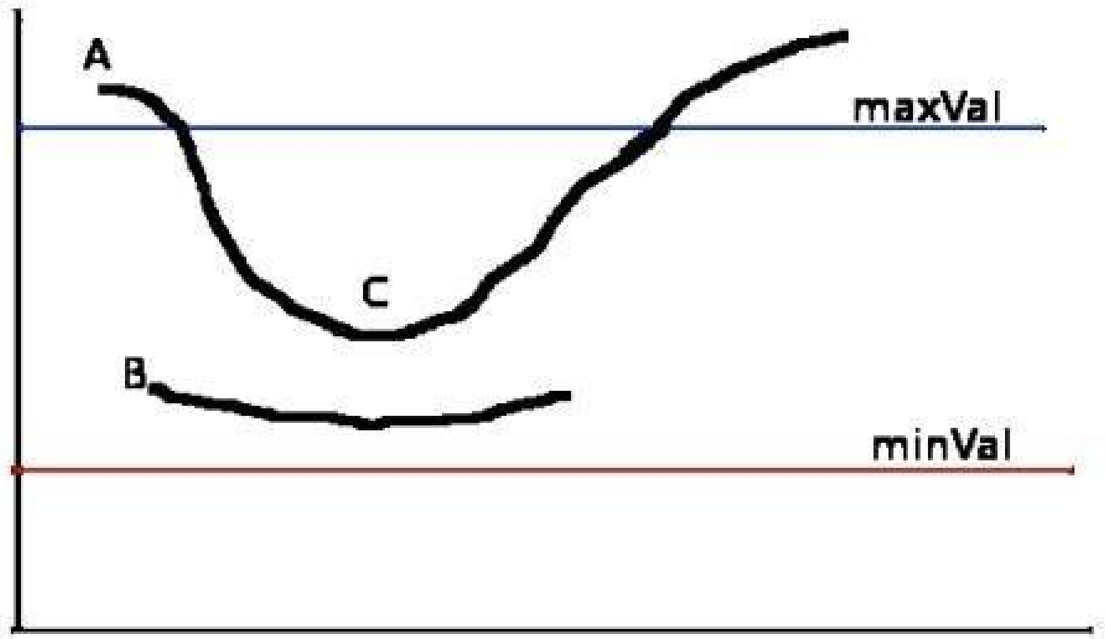


Point A is on the edge (in vertical direction). Gradient direction is normal to the edge. Point B and C are in gradient directions. So point A is checked with point B and C to see if it forms a local maximum. If so, it is considered for next stage, otherwise, it is suppressed (put to zero).

In short, the result you get is a binary image with “thin edges”.

4. Hysteresis Thresholding

This stage decides which are all edges are really edges and which are not. For this, we need two threshold values, *minVal* and *maxVal*. Any edges with intensity gradient more than *maxVal* are sure to be edges and those below *minVal* are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to “sure-edge” pixels, they are considered to be part of edges. Otherwise, they are also discarded. See the image below:



The edge A is above the *maxVal*, so considered as “sure-edge”. Although edge C is below *maxVal*, it is connected to edge A, so that also considered as valid edge and we get that full curve. But edge B, although it is above *minVal* and is in same region as that of edge C, it is not connected to any “sure-edge”, so that is discarded. So it is very important that we have to select *minVal* and *maxVal* accordingly to get the correct result.

This stage also removes small pixels noises on the assumption that edges are long lines.

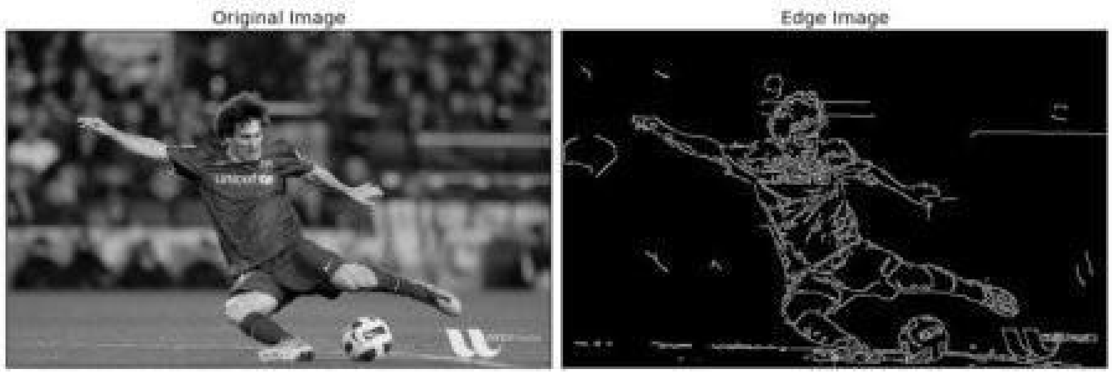
So what we finally get is strong edges in the image.

Canny Edge Detection in OpenCV

OpenCV puts all the above in single function, cv2.Canny(). We will see how to use it. First argument is our input image. Second and third arguments are our *minVal* and *maxVal* respectively. Third argument is *aperture_size*. It is the size of Sobel kernel used for find image gradients. By default it is 3. Last argument is *L2gradient* which specifies the equation for finding gradient magnitude. If it is True, it uses the equation mentioned above which is more accurate, otherwise it uses this function: $Edge\_Gradient\ (G)$ = $|G_x|$ + $|G_y|$. By default, it is False.

```
import cv2 import numpy
as np
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg',0) edges =
cv2.Canny(img,100,200)

plt.subplot(121),plt.imshow(img,cmap = 'gray') plt.title('Original Image'), plt.xticks([]),
plt.yticks([]) plt.subplot(122),plt.imshow(edges,cmap = 'gray') plt.title('Edge Image'),
plt.xticks([]), plt.yticks([])

plt.show()
```

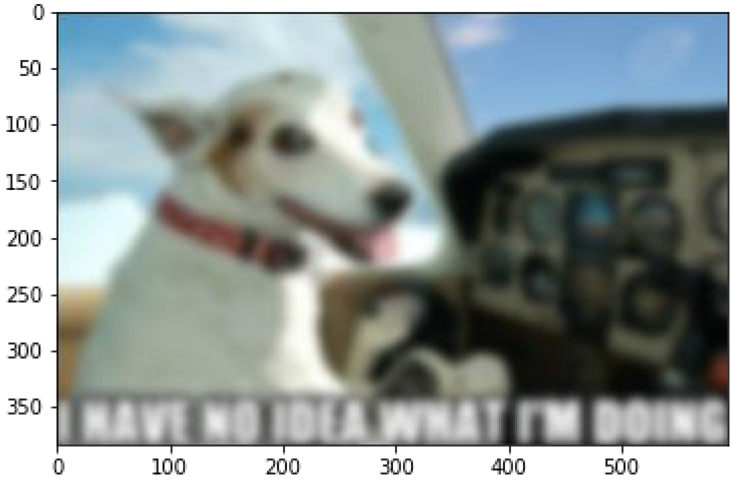
See the result below:



Exercises

1. Write a small application to find the Canny edge detection whose threshold values can be varied using two trackbars. This way, you can understand the effect of threshold values.

Colab Code



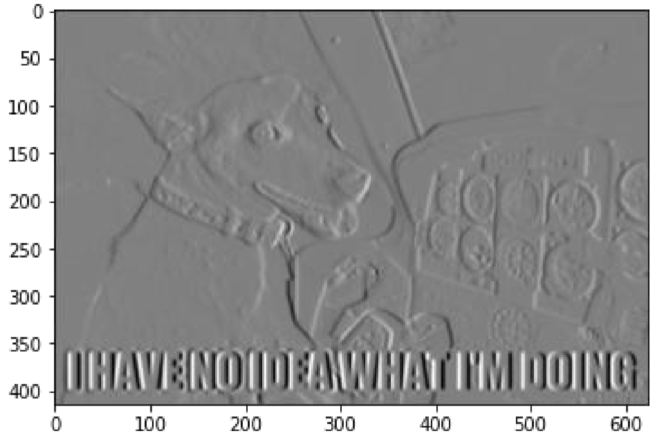
Edges

Edge detection is the final image processing technique we're going to look at in this tutorial.

For a lot of what we think of as "modern" computer vision techniques, edge detection functions as a building block. Much edge detection actually works by convolution, and indeed convolutional neural networks are absolutely the flavor of the month in some parts of computer vision. Sobel's edge detector was one of the first truly successful edge detection (enhancement) techniques and that involves convolution at its core.

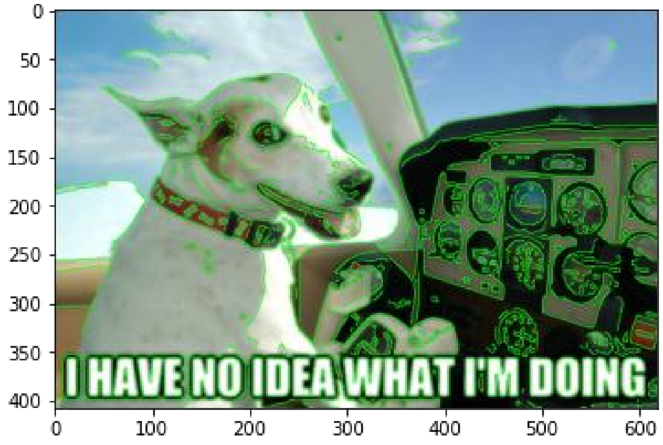
```
sobelimage=cv2.cvtColor(input_image,cv2.COLOR_BGR2GRAY)
sobelx = cv2.Sobel(sobelimage,cv2.CV_64F,1,0,ksize=9)
sobely = cv2.Sobel(sobelimage,cv2.CV_64F,0,1,ksize=9)
```

```
plt.imshow(sobelx,cmap·=·'gray')·
#·Sobel·works·in·x·and·in·y,·change·sobelx·to·sobely·in·the·olt·line·above
·to·see·the·difference
```



Canny edge detection is another winning technique - it takes two thresholds. The first one determines how likely Canny is to find an edge, and the second determines how likely it is to follow that edge once it's found. Investigate the effect of these thresholds by altering the values below.

```
th1=30
th2=60 # Canny recommends threshold 2 is 3 times threshold 1 - you could
try experimenting with this...
d=3 # gaussian blur
edgeresult=input_image.copy()
edgeresult = cv2.GaussianBlur(edgeresult, (2*d+1, 2*d+1), -1)[d:-d,d:-d]
gray = cv2.cvtColor(edgeresult, cv2.COLOR_BGR2GRAY)
edge = cv2.Canny(gray, th1, th2)
edgeresult[edge != 0] = (0, 255, 0) # this takes pixels in edgeresult
where edge non-zero colors them bright green
plt.imshow(cv2.cvtColor(edgeresult, cv2.COLOR_BGR2RGB))
```



Exercises

1. Implement a simple python program with an image of your choice and demonstrate all the concepts (attach screenshots).

LAB 9

Histograms - 1 : Find, Plot, Analyze !!!

Goal

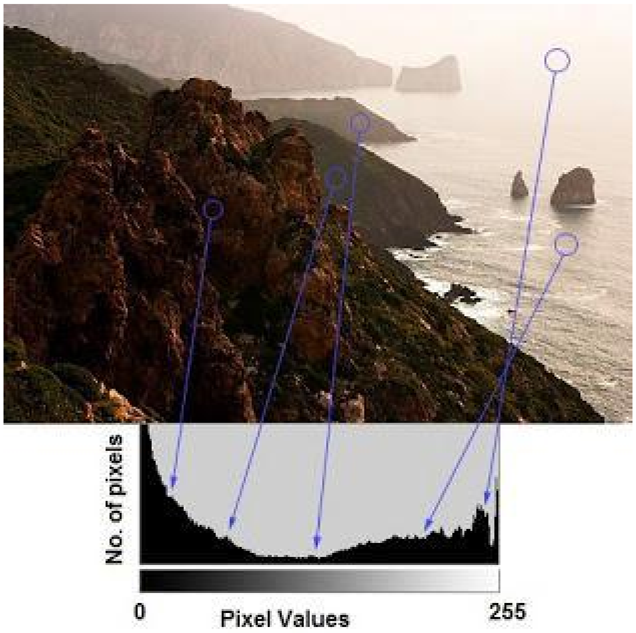
Learn to

- Find histograms, using both OpenCV and Numpy functions
- Plot histograms, using OpenCV and Matplotlib functions
- You will see these functions : **cv.calcHist()**, **np.histogram()** etc.

Theory

So what is histogram ? You can consider histogram as a graph or plot, which gives you an overall idea about the intensity distribution of an image. It is a plot with pixel values (ranging from 0 to 255, not always) in X-axis and corresponding number of pixels in the image on Y-axis.

It is just another way of understanding the image. By looking at the histogram of an image, you get intuition about contrast, brightness, intensity distribution etc of that image. Almost all image processing tools today, provides features on histogram. Below is an image from Cambridge in Color website, and I recommend you to visit the site for more details.



image

You can see the image and its histogram. (Remember, this histogram is drawn for grayscale image, not color image). Left region of histogram shows the amount of darker pixels in image and right region shows the amount of brighter pixels. From the histogram, you can see dark region is more than brighter region, and amount of midtones (pixel values in mid-range, say around 127) are very less.

Find Histogram

Now we have an idea on what is histogram, we can look into how to find this. Both OpenCV and Numpy come with in-built function for this. Before using those functions, we need to understand some terminologies related with histograms.

BINS :The above histogram shows the number of pixels for every pixel value, ie from 0 to 255. ie you need 256 values to show the above histogram. But consider, what if you need not find the number of pixels for all pixel values separately, but number of pixels in a interval of pixel values? say for example, you need to find the number of pixels lying between 0 to 15, then 16 to 31, ..., 240 to 255. You will need only 16 values to represent the histogram. And that is what is shown in example given in **OpenCV Tutorials on histograms**.

So what you do is simply split the whole histogram to 16 sub-parts and value of each sub-part is the sum of all pixel count in it. This each sub-part is called "BIN". In first case, number of bins were 256 (one for each pixel) while in second case, it is only 16. BINS is represented by the term **histSize** in OpenCV docs.

DIMS : It is the number of parameters for which we collect the data. In this case, we collect data regarding only one thing, intensity value. So here it is 1.

RANGE : It is the range of intensity values you want to measure. Normally, it is [0,256], ie all intensity values.

1. Histogram Calculation in OpenCV

So now we use **cv.calcHist()** function to find the histogram. Let's familiarize with the function and its parameters :

cv.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])

1. images : it is the source image of type uint8 or float32. it should be given in square brackets, ie, "[img]".
2. channels : it is also given in square brackets. It is the index of channel for which we calculate histogram. For example, if input is grayscale image, its value is [0]. For color image, you can pass [0], [1] or [2] to calculate histogram of blue, green or red channel respectively.
3. mask : mask image. To find histogram of full image, it is given as "None". But if you want to find histogram of particular region of image, you have to create a mask image for that and give it as mask. (I will show an example later.)
4. histSize : this represents our BIN count. Need to be given in square brackets. For full scale, we pass [256].
5. ranges : this is our RANGE. Normally, it is [0,256].

So let's start with a sample image. Simply load an image in grayscale mode and find its full histogram.

```
img = cv.imread('home.jpg',0)
hist = cv.calcHist([img],[0],None,[256],[0,256])
```

hist is a 256x1 array, each value corresponds to number of pixels in that image with its corresponding pixel value.

2. Histogram Calculation in Numpy

Numpy also provides you a function, **np.histogram()**. So instead of **calcHist()** function, you can try below line :

```
hist,bins = np.histogram(img.ravel(),256,[0,256])
```

hist is same as we calculated before. But bins will have 257 elements, because Numpy calculates bins as 0-0.99, 1-1.99, 2-2.99 etc. So final range would be 255-255.99. To represent that, they also add 256 at end of bins. But we don't need that 256. Upto 255 is sufficient.

Note

> Numpy has another function, **np.bincount()** which is much faster than (around 10X) np.histogram(). So for one-dimensional histograms, you can better try that. Don't forget to set minlength = 256 in np.bincount. For example, hist = np.bincount(img.ravel(),minlength=256)
>
> OpenCV function is faster than (around 40X) than np.histogram(). So stick with OpenCV function.

Now we should plot histograms, but how?

Plotting Histograms

There are two ways for this,

1. Short Way : use Matplotlib plotting functions
2. Long Way : use OpenCV drawing functions

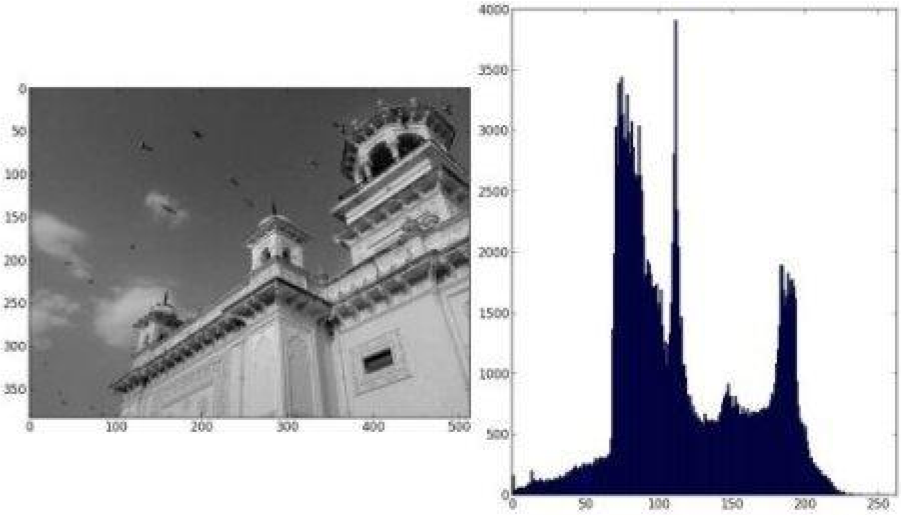
1. Using Matplotlib

Matplotlib comes with a histogram plotting function : matplotlib.pyplot.hist()

It directly finds the histogram and plot it. You need not use **calcHist()** or np.histogram() function to find the histogram. See the code below:

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('home.jpg',0)
plt.hist(img.ravel(),256,[0,256]); plt.show()
```

You will get a plot as below :

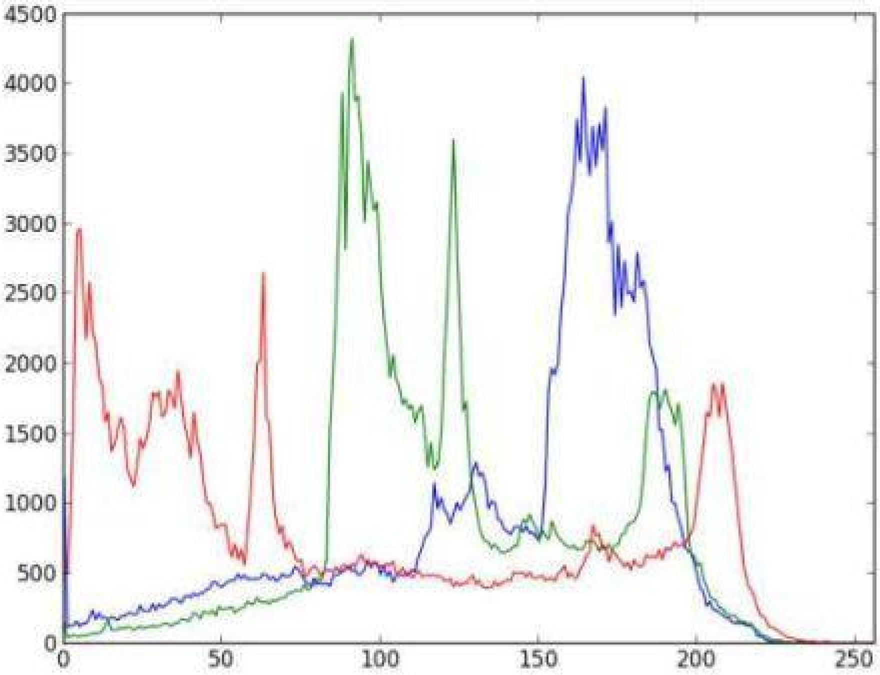


image

Or you can use normal plot of matplotlib, which would be good for BGR plot. For that, you need to find the histogram data first. Try below code:

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('home.jpg')
color = ('b','g','r')
for i,col in enumerate(color):
histr = cv.calcHist([img],[i],None,[256],[0,256])
plt.plot(histr,color = col)
plt.xlim([0,256])
plt.show()
```

Result:



image

You can deduct from the above graph that, blue has some high value areas in the image (obviously it should be due to the sky)

2. Using OpenCV

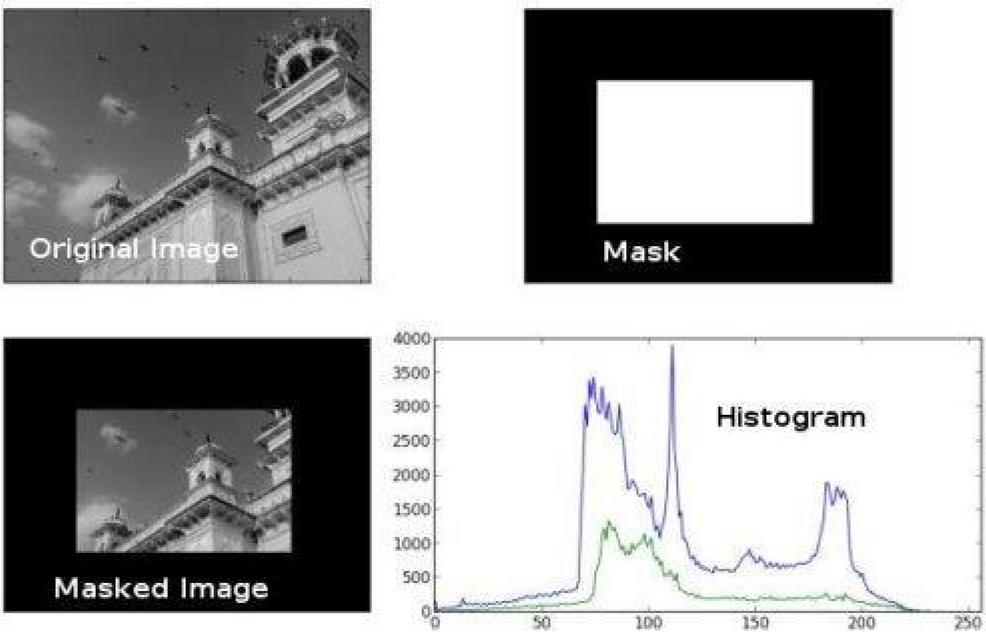
Well, here you adjust the values of histograms along with its bin values to look like x,y coordinates so that you can draw it using **cv.line()** or cv.polyline() function to generate same image as above. This is already available with OpenCV-Python2 official samples. Check the code at samples/python/hist.py.

Application of Mask

We used **cv.calcHist()** to find the histogram of the full image. What if you want to find histograms of some regions of an image? Just create a mask image with white color on the region you want to find histogram and black otherwise. Then pass this as the mask.

```
img = cv.imread('home.jpg',0)
# create a mask
mask = np.zeros(img.shape[:2], np.uint8)
mask[100:300, 100:400] = 255
masked_img = cv.bitwise_and(img,img,mask = mask)
# Calculate histogram with mask and without mask
# Check third argument for mask
hist_full = cv.calcHist([img],[0],None,[256],[0,256])
hist_mask = cv.calcHist([img],[0],mask,[256],[0,256])
plt.subplot(221), plt.imshow(img, 'gray')
plt.subplot(222), plt.imshow(mask,'gray')
plt.subplot(223), plt.imshow(masked_img, 'gray')
plt.subplot(224), plt.plot(hist_full), plt.plot(hist_mask)
plt.xlim([0,256])
plt.show()
```

See the result. In the histogram plot, blue line shows histogram of full image while green line shows histogram of masked region.



image

LAB 10

histogram equalization

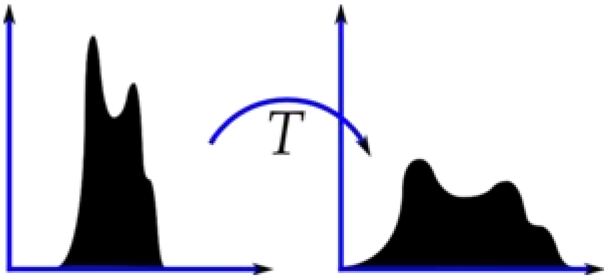
Goal

In this section,

- We will learn the concepts of histogram equalization and use it to improve the contrast of our images.

Theory

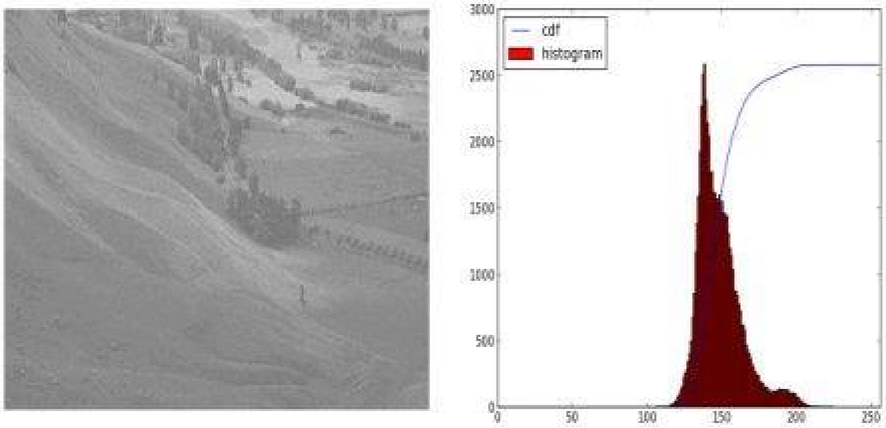
Consider an image whose pixel values are confined to some specific range of values only. For eg, brighter image will have all pixels confined to high values. But a good image will have pixels from all regions of the image. So you need to stretch this histogram to either ends (as given in below image, from wikipedia) and that is what Histogram Equalization does (in simple words). This normally improves the contrast of the image.



image

I would recommend you to read the wikipedia page on Histogram Equalization for more details about it. It has a very good explanation with worked out examples, so that you would understand almost everything after reading that. Instead, here we will see its Numpy implementation. After that, we will see OpenCV function.

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('wiki.jpg',0)
hist,bins = np.histogram(img.flatten(),256,[0,256])
cdf = hist.cumsum()
cdf_normalized = cdf * float(hist.max()) / cdf.max()
plt.plot(cdf_normalized, color = 'b')
plt.hist(img.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(('cdf','histogram'), loc = 'upper left')
plt.show()
```



image

You can see histogram lies in brighter region. We need the full spectrum. For that, we need a transformation function which maps the input pixels in brighter region to output pixels in full region. That is what histogram equalization does.

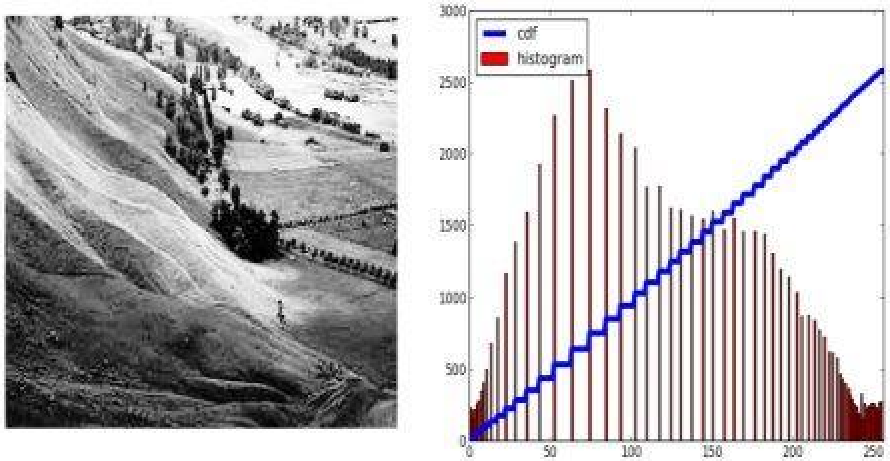
Now we find the minimum histogram value (excluding 0) and apply the histogram equalization equation as given in wiki page. But I have used here, the masked array concept array from Numpy. For masked array, all operations are performed on non-masked elements. You can read more about it from Numpy docs on masked arrays.

```
cdf_m = np.ma.masked_equal(cdf,0)
cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
cdf = np.ma.filled(cdf_m,0).astype('uint8')
```

Now we have the look-up table that gives us the information on what is the output pixel value for every input pixel value. So we just apply the transform.

```
img2 = cdf[img]
```

Now we calculate its histogram and cdf as before (you do it) and result looks like below :



image

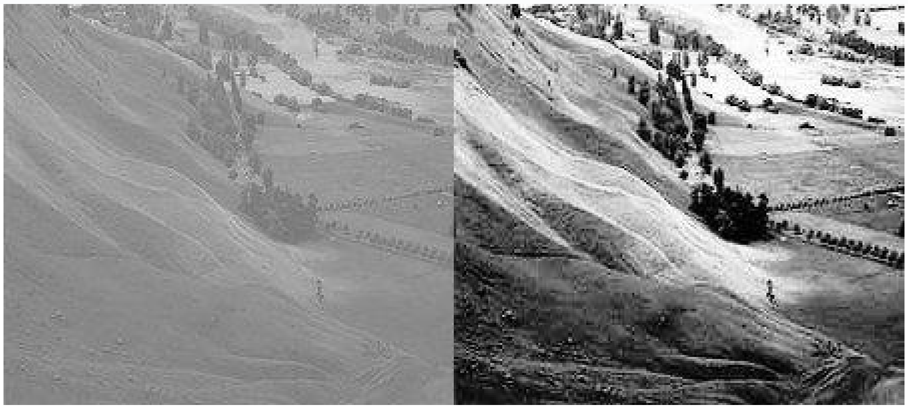
Another important feature is that, even if the image was a darker image (instead of a brighter one we used), after equalization we will get almost the same image as we got. As a result, this is used as a "reference tool" to make all images with same lighting conditions. This is useful in many cases. For example, in face recognition, before training the face data, the images of faces are histogram equalized to make them all with same lighting conditions.

Histograms Equalization in OpenCV

OpenCV has a function to do this, **cv.equalizeHist()**. Its input is just grayscale image and output is our histogram equalized image.

Below is a simple code snippet showing its usage for same image we used :

```
img = cv.imread('wiki.jpg',0)
equ = cv.equalizeHist(img)
res = np.hstack((img,equ)) #stacking images side-by-side
cv.imwrite('res.png',res)
```



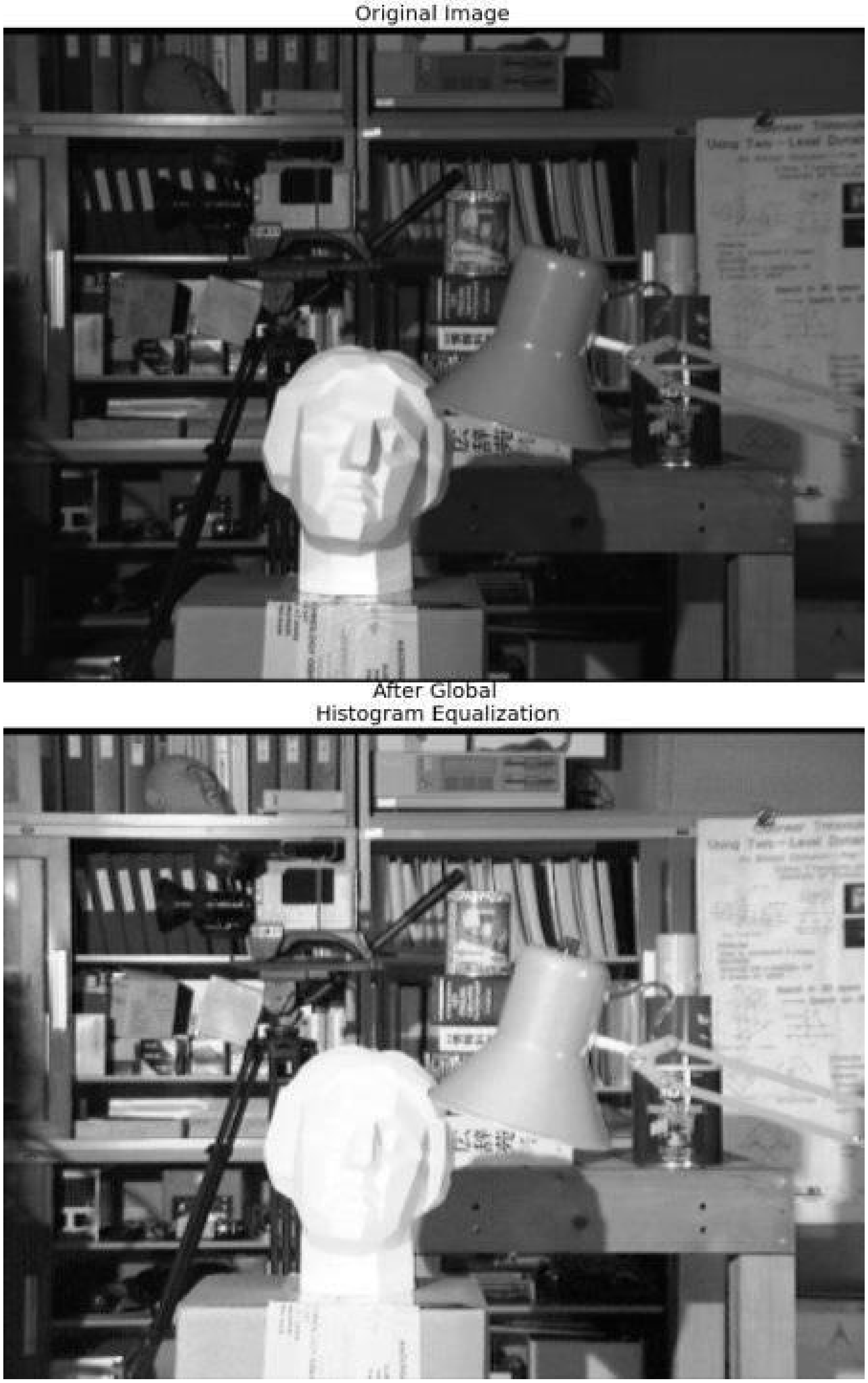
image

So now you can take different images with different light conditions, equalize it and check the results.

Histogram equalization is good when histogram of the image is confined to a particular region. It won't work good in places where there is large intensity variations where histogram covers a large region, ie both bright and dark pixels are present. Please check the SOF links in Additional Resources.

CLAHE (Contrast Limited Adaptive Histogram Equalization)

The first histogram equalization we just saw, considers the global contrast of the image. In many cases, it is not a good idea. For example, below image shows an input image and its result after global histogram equalization.



Original Image

After Global Histogram Equalization

image

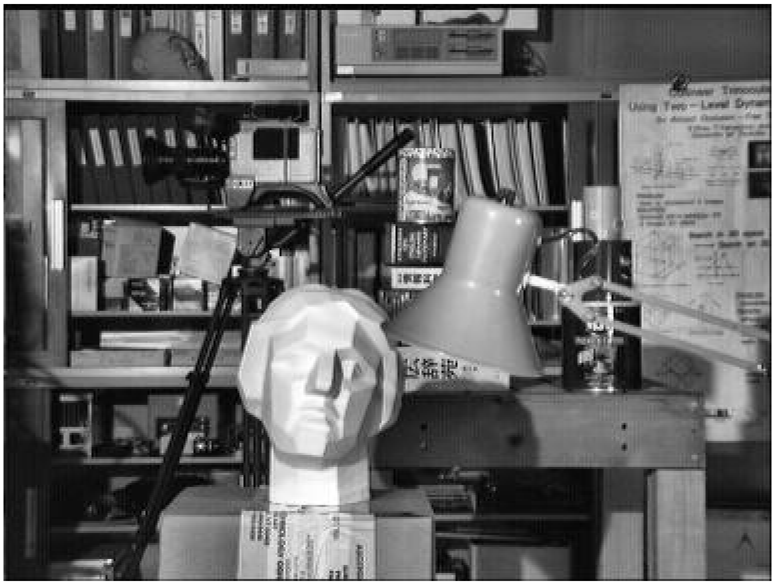
It is true that the background contrast has improved after histogram equalization. But compare the face of statue in both images. We lost most of the information there due to over-brightness. It is because its histogram is not confined to a particular region as we saw in previous cases (Try to plot histogram of input image, you will get more intuition).

So to solve this problem, **adaptive histogram equalization** is used. In this, image is divided into small blocks called "tiles" (tileSize is 8x8 by default in OpenCV). Then each of these blocks are histogram equalized as usual. So in a small area, histogram would confine to a small region (unless there is noise). If noise is there, it will be amplified. To avoid this, **contrast limiting** is applied. If any histogram bin is above the specified contrast limit (by default 40 in OpenCV), those pixels are clipped and distributed uniformly to other bins before applying histogram equalization. After equalization, to remove artifacts in tile borders, bilinear interpolation is applied.

Below code snippet shows how to apply CLAHE in OpenCV:

```
import numpy as np
import cv2 as cv
img = cv.imread('tsukuba_l.png',0)
# create a CLAHE object (Arguments are optional).
clahe = cv.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
cl1 = clahe.apply(img)
cv.imwrite('clahe_2.jpg',cl1)
```

See the result below and compare it with results above, especially the statue region:



Lab 11

Features in computer vision

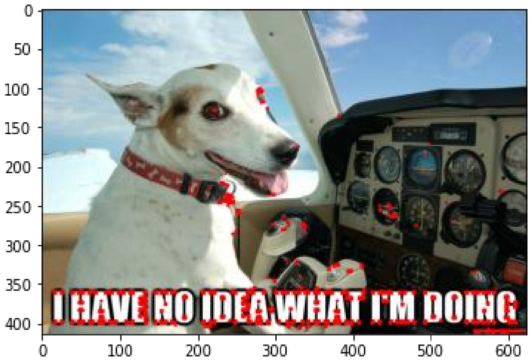
Features are image locations that are "easy" to find in the future. Indeed, one of the early feature detection techniques Lucas-Kanade, sometimes called Kanade-Lucas-Tomasi or KLT features, comes from a seminal paper called "Good features to track".
Edges find brightness discontinuities in an image, features find distinctive regions. There are a bunch of different feature detectors and these all have some characteristics in common: they should be quick to find, and things that are close in image-space are close in feature-space (that is, the feature representation of an object looks like the feature representation of objects that look like that object).

Corner detectors

If you think of edges as being lines, then corners are an obvious choice for features as they represent the intersection of two lines. One of the earlier corner detectors was introduced by Harris, and it is still a very effective corner detector that gets used quite a lot: it's reliable and it's fast.

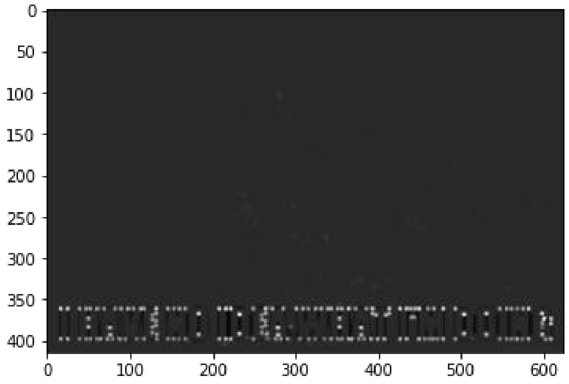
harris_test=input_image.copy()

```
    #greyscale it
    gray = cv2.cvtColor(harris_test,cv2.COLOR_BGR2GRAY)
    gray = np.float32(gray)
    blocksize=4 #
    kernel_size=3 # sobel kernel: must be odd and fairly small
    # run the harris corner detector
    dst = cv2.cornerHarris(gray,blocksize,kernel_size,0.05) # parameters are
    blocksize, Sobel parameter and Harris threshold
    #result is dilated for marking the corners, this is visualisation related
    and just makes them bigger
    dst = cv2.dilate(dst,None)
    #we then plot these on the input image for visualization purposes, using
    bright red
    harris_test[dst>0.01*dst.max()]=[0,0,255]
    plt.imshow(cv2.cvtColor(harris_test, cv2.COLOR_BGR2RGB))
```



Properly speaking the Harris Corner detection is more like a Sobel operator - indeed it is very much like a sobel operator. It doesn't really return a set of features, instead it is a filter which gives a strong response on corner-like regions of the image. We can see this more clearly if we look at the Harris output from the cell above (dst is the Harris response, before thresholding). Well we can kind-of see. You should be able to see that there are slightly light places in the image where there are corner like features, and that there are really light parts of the image around the black and white corners of the writing

```
plt.imshow(dst,cmap = 'gray')
```



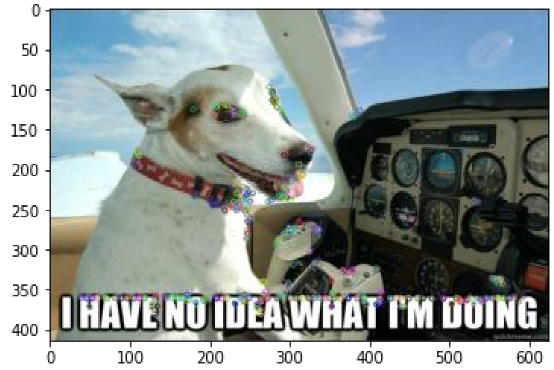
Moving towards feature space

When we consider modern feature detectors there are a few things we need to mention. What makes a good feature includes the following:

- Repeatability (got to be able to find it again)
- Distinctiveness/informativeness (features representing different things need to be different)
- Locality (they need to be local to the image feature and not, like, the whole image)
- Quantity (you need to be able to find enough of them for them to be properly useful)
- Accuracy (they need to accurately locate the image feature)
- Efficiency (they've got to be computable in reasonable time)

Note: some of the very famous feature detectors (SIFT/SURF and so on) are around, but aren't in OpenCV by default due to patent issues. You can build them for OpenCV if you want - or you can find other implementations (David Lowe's SIFT implementation works just fine). Just google for instructions. For the purposes of this tutorial (and to save time) we're only going to look at those which are actually in OpenCV.

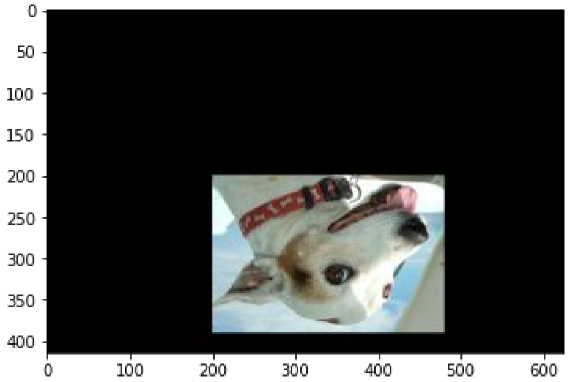
```
orbimg=input_image.copy()
orb = cv2.ORB_create()
# find the keypoints with ORB
kp = orb.detect(orbimg,None)
# compute the descriptors with ORB
kp, des = orb.compute(orbimg, kp)
# draw keypoints
cv2.drawKeypoints(orbimg,kp,orbimg)
plt.imshow(cv2.cvtColor(orbimg, cv2.COLOR_BGR2RGB))
```



Matching features

Finding features is one thing but actually we want to use them for matching. First let's get

```
img2match=np.zeros(input_image.shape,np.uint8)
dogface=input_image[60:250, 70:350] # copy out a bit
img2match[60:250,70:350]=[0,0,0] # blank that region
dogface=cv2.flip(dogface,0) #flip the copy
img2match[200:200+dogface.shape[0], 200:200+dogface.shape[1]]=dogface
# paste it back somewhere else
plt.imshow(cv2.cvtColor(img2match, cv2.COLOR_BGR2RGB))
```

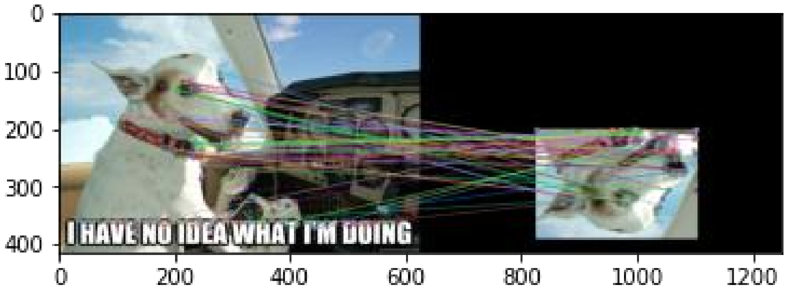


Matching keypoints

The feature matching function (in this case Orb) detects and then computes keypoint descriptors. These are a higher dimensional representation of the image region immediately around a point of interest (sometimes literally called "interest points").

These higher-dimensional representations can then be matched; the strength you gain from matching these descriptors rather than image regions directly is that they have a certain invariant to transformations (like rotation, or scaling). OpenCV providers match routines to do this, in which you can specify the distance measure to use.

```
kp2 = orb.detect(img2match,None)
# compute the descriptors with ORB
kp2, des2 = orb.compute(img2match, kp2)
# create BFMatcher object: this is a Brute Force matching object
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
# Match descriptors.
matches = bf.match(des,des2)
# Sort them by distance between matches in feature space
# so the best matches are first.
matches = sorted(matches, key = lambda x:x.distance)
# Draw first 50 matches.
oimg = cv2.drawMatches(orbimg,kp,img2match,kp2,matches[:50], orbimg)
plt.imshow(cv2.cvtColor(oimg, cv2.COLOR_BGR2RGB))
```



As you can see there are some false matches, but it's fairly clear that most of the matched keypoints found are actual matches between image regions on the dogface.

To be more precise about our matching we could choose to enforce homography constraints, which looks for features rather than sit on the same plane.

Exercises

1. Implement a simple python program with an image of your choice and demonstrate all the concepts (attach screenshots).

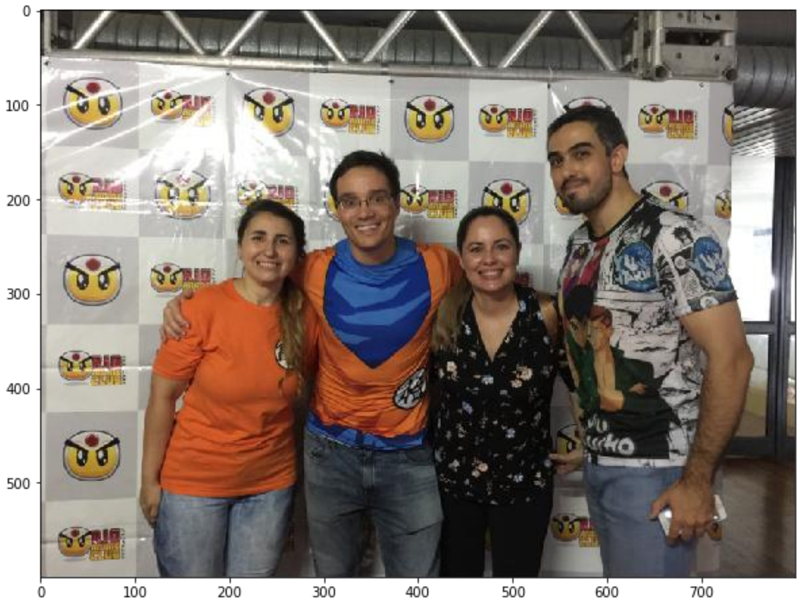
Lab 12
Cascade Classification

One of the key things we can do with vision is object detection.

Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper and improved later by Rainer Lienhart. 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.

OpenCV provides a training method or pretrained models that can be read using the cv2.CascadeClassifierload method. In this lab we will play with some of the provided pre-trained haarcascades models. Load the test image and create a grayscale copy of it to be used in the classifiers

```
base_image = cv2.imread('test.jpg')
gray = cv2.cvtColor(base_image, cv2.COLOR_BGR2GRAY)
plt.imshow(cv2.cvtColor(base_image, cv2.COLOR_BGR2RGB))
```



Face Detection

We will use the pre-trained model haarcascade_frontalface_default.xml to detect faces in the photo. You can find more details about the parameters of the detectMultiScale function here.

Note: In all examples, I reload the color image again, because the imshow function rewrites the original image with the boxes, but I use the same grayscale image for detection)

```
# this is a pre-trained face cascade
test_image = cv2.imread('test.jpg')
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
```

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
     cv2.rectangle(test_image,(x,y),(x+w,y+h),(255,0,0),2)
plt.imshow(cv2.cvtColor(test_image, cv2.COLOR_BGR2RGB))
```

Smile Detection

We will use the pre-trained model haarcascade_smile.xml to detect smiles on the pictures.

```
# this is a pre-trained face cascade
test_image = cv2.imread('test.jpg')
smile_cascade = cv2.CascadeClassifier('haarcascade_smile.xml')
smiles = smile_cascade.detectMultiScale(gray, 1.3, 20)
for (x,y,w,h) in smiles:
     cv2.rectangle(test_image,(x,y),(x+w,y+h),(0,255,0),2)
plt.imshow(cv2.cvtColor(test_image, cv2.COLOR_BGR2RGB))
```

As you can see, it detected the smiles correctly (the right guy, who is actually me, is not giving a full smile) but there are a lot of false positives, as usual in cascade models. Therefore, to improve this, we will only consider detected smiles inside previously detected faces.

```
# this is a pre-trained face cascade
test_image = cv2.imread('test.jpg')
for (x,y,w,h) in faces:
  for (x_s,y_s,w_s,h_s) in smiles:
    if( (x <= x_s) and (y <= y_s) and ( x+w >= x_s+w_s) and ( y+h >=
y_s+h_s)):
      cv2.rectangle(test_image, (x_s,y_s),(x_s+w_s,y_s+h_s),(0,255,0),2)
plt.imshow(cv2.cvtColor(test_image, cv2.COLOR_BGR2RGB))
```

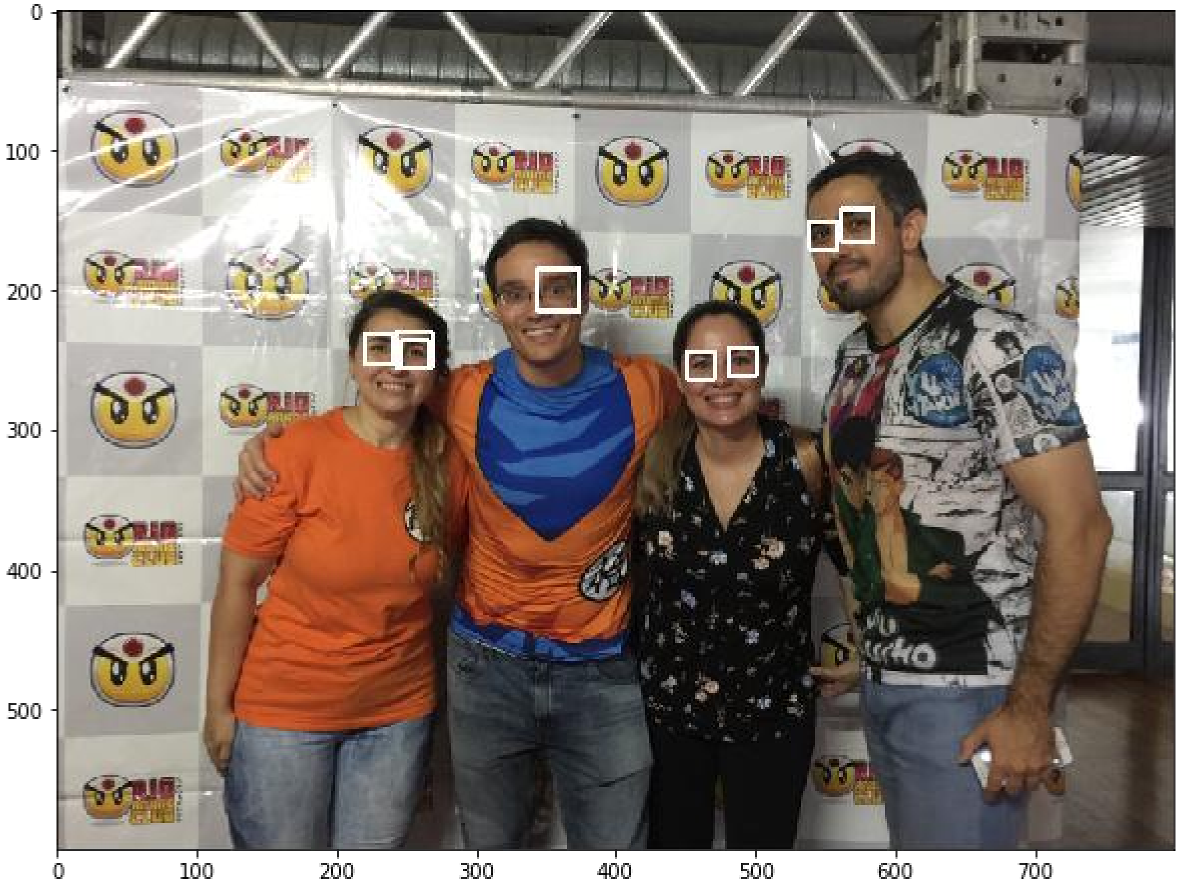
Eye Detection

Using the pre-trained model haarcascade_frontalface_default.xml to detect faces on the pictures.

```
test_image = cv2.imread('test.jpg')
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')
eyes = eye_cascade.detectMultiScale(gray, 1.3, 1)
for (x,y,w,h) in eyes:
     cv2.rectangle(test_image,(x,y),(x+w,y+h),(255,255,255),2)
plt.imshow(cv2.cvtColor(test_image, cv2.COLOR_BGR2RGB))
```

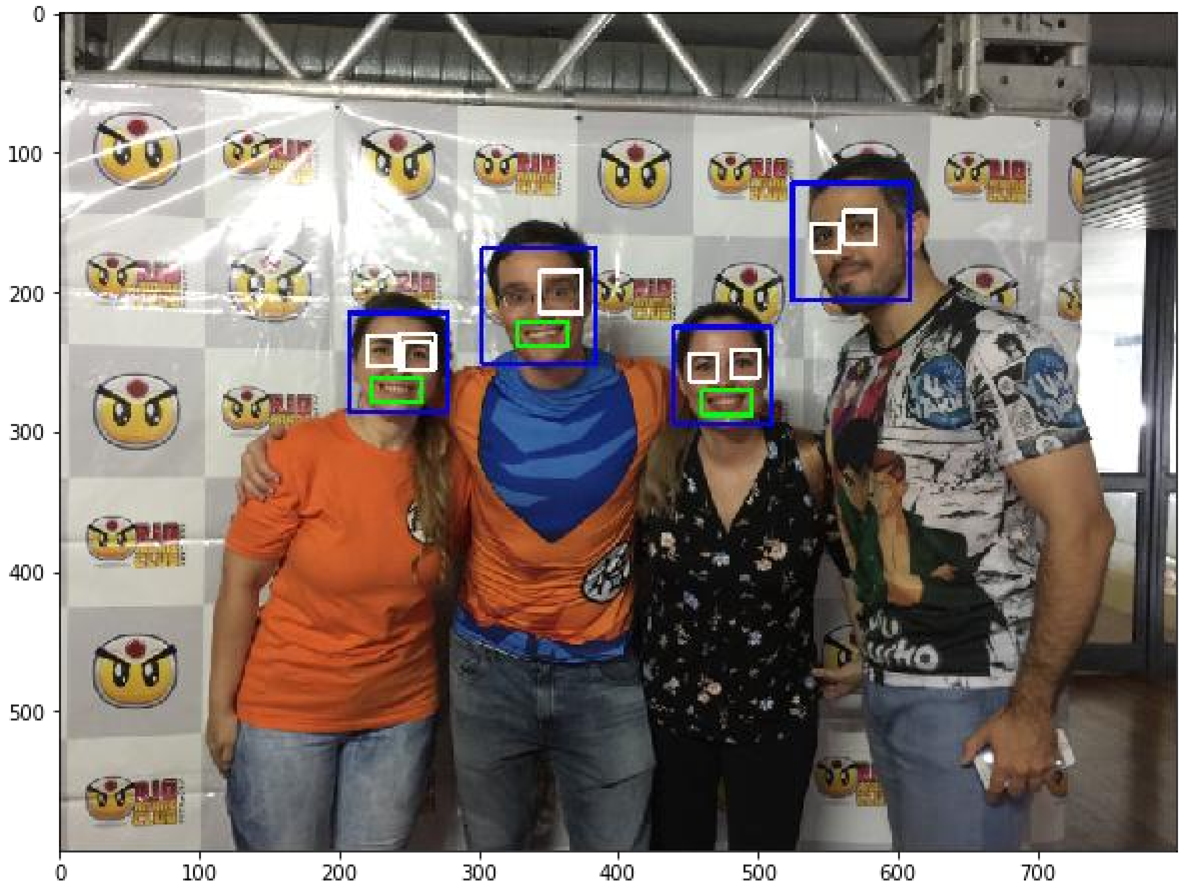
Similar to what happened to the smiles, there are a few false positives, and in this particular case a false negative (probably the glass confused the classifier). So we will use the same approach to filter recognized eyes that are inside faces.

```
      test_image = cv2.imread('test.jpg')
      for (x,y,w,h) in faces:
        #cv2.rectangle(smile_faces_base_image,(x,y),(x+w,y+h),(255,0,0),2)
        for (x_s,y_s,w_s,h_s) in eyes:
          if( (x <= x_s) and (y <= y_s) and ( x+w >= x_s+w_s) and ( y+h >=
y_s+h_s)):
            cv2.rectangle(test_image,
(x_s,y_s),(x_s+w_s,y_s+h_s),(255,255,255),2)
      plt.imshow(cv2.cvtColor(test_image, cv2.COLOR_BGR2RGB))
```



Putting all together

```
      test_image = cv2.imread('test.jpg')
      for (x,y,w,h) in faces:
        cv2.rectangle(test_image,(x,y),(x+w,y+h),(255,0,0),2)
        for (x_s,y_s,w_s,h_s) in eyes:
          if( (x <= x_s) and (y <= y_s) and ( x+w >= x_s+w_s) and ( y+h >=
y_s+h_s)):
            cv2.rectangle(test_image,
(x_s,y_s),(x_s+w_s,y_s+h_s),(255,255,255),2)
        for (x_s,y_s,w_s,h_s) in smiles:
          if( (x <= x_s) and (y <= y_s) and ( x+w >= x_s+w_s) and ( y+h >=
y_s+h_s)):
            cv2.rectangle(test_image, (x_s,y_s),(x_s+w_s,y_s+h_s),(0,255,0),2)
      plt.imshow(cv2.cvtColor(test_image, cv2.COLOR_BGR2RGB))
```



Modern Object Recognition

Although most of this type of object detection has recently been replaced in the industry by solutions based on Deep Neural Networks models created and trained in platforms like TensorFlow, the Viola Jones method for face detection is still a really well respected detector. And even in modern DNN-based systems OpenCV is still widely used for image manipulation, pre and post-processing .

Exercises

1. Implement a simple python program with an image of your choice and demonstrate all the concepts (attach screenshots).