OpenCV

Prerequisite: Before starting this exercise, you should make yourself familiar with Python and some necessary library, e.g., numpy, matplotlib, etc. One good tutorial can be found here.

In this exercise you will:

- Learn about some basic image processing operations with OpenCV.
- Re-implement some basic image processing operations. This will help you to
- Have better understand about the image processing operations.
- Practice Python programming with Numpy library.

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

# This is a bit of magic to make matplotlib figures appear inline in
the notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of
plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

def rel_error(out, correct_out):
    return np.sum(abs(out.astype(np.float32) -
correct_out.astype(np.float32)) /
                         (abs(out.astype(np.float32)) +
abs(correct_out.astype(np.float32))))

# !pip install opencv-python==3.4.17.61

# Checking OpenCV version
cv2.__version__
```

```
'3.4.17'
```

NOTICE:

In this lab exercise, we recommend to use `OpenCV 3.x` version, the documentations for OpenCV API can be found here.

Load images

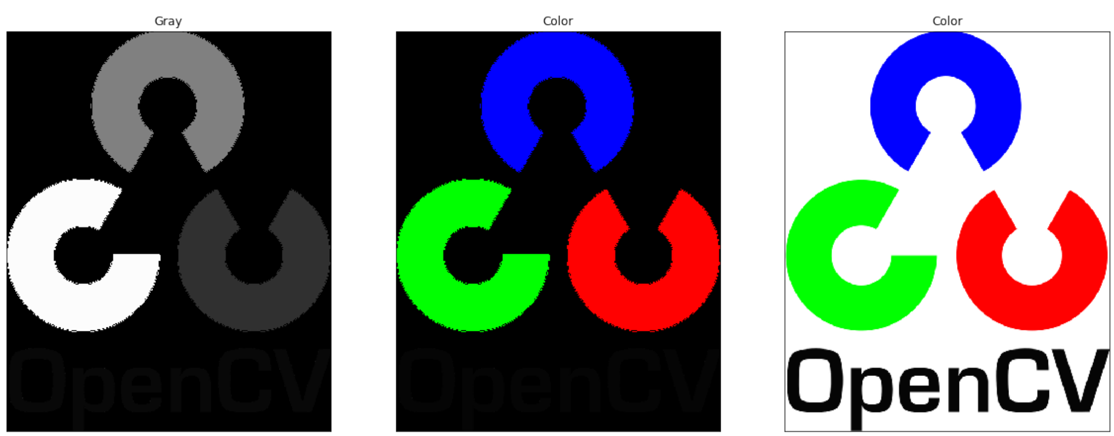
Use the function cv2.imread() to read an image. The image should be in the working directory or a full path of image should be given. The function will return a numpy matrix.

Second argument is a flag which specifies the way image should be read.

- cv2.IMREAD_COLOR - (1): Loads a color image. Any transparency (alpha channel) of image will be neglected. It is the **default flag**.
- cv2.IMREAD_GRAYSCALE - (0): Loads image in grayscale mode
- cv2.IMREAD_UNCHANGED - (-1): Loads image as such including alpha channel, if included.

NOTE: Color image loaded by OpenCV is in *Blue-Green-Red (BGR)* mode. But Matplotlib displays in *RGB* mode. So color images will not be displayed correctly in Matplotlib if image is read with OpenCV. We will discuss how to handle to display properly later.

```
img_gray = cv2.imread('imgs/opencv_logo.png', 0)

plt.figure(figsize=(20,10))
plt.subplot(131),
plt.imshow(img_gray, cmap='gray')  # include cmap='gray' to display gray image
plt.title('Gray'),plt.xticks([]), plt.yticks([])

img_color1= cv2.imread('imgs/opencv_logo.png', 1)
plt.subplot(132),plt.imshow(img_color1),
plt.title('Color'),plt.xticks([]), plt.yticks([])

img_color2= cv2.imread('imgs/opencv_logo.png',-1)
plt.subplot(133),plt.imshow(img_color2),
plt.title('Color'),plt.xticks([]), plt.yticks([])
plt.show()
```



Question: How many channels for each image: img_gray, img_color1, img_color2?

Your answer:

- img_gray: 1 (Grayscale)
- img_color1: 3 (RGB)
- img_color2: 4 (RGB + Alpha)

Transformations

Scaling

Resize image using the function cv2.resize.

```
# Get list of available flags
flags = [i for i in dir(cv2) if i.startswith('INTER_')]
print (flags)
```

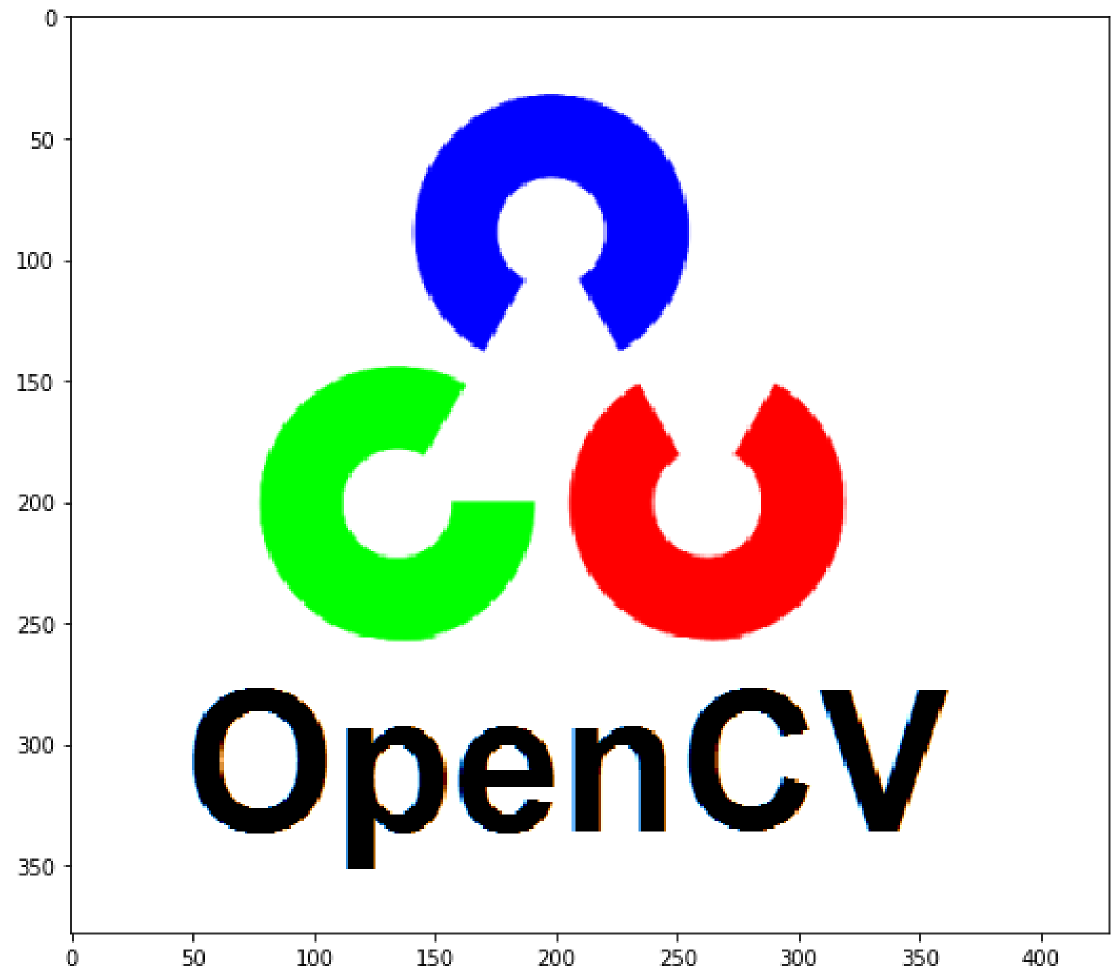
```
['INTER_AREA', 'INTER_BITS', 'INTER_BITS2', 'INTER_CUBIC',
'INTER_LANCZOS4', 'INTER_LINEAR', 'INTER_LINEAR_EXACT', 'INTER_MAX',
'INTER_NEAREST', 'INTER_NEAREST_EXACT', 'INTER_TAB_SIZE',
'INTER_TAB_SIZE2']
```

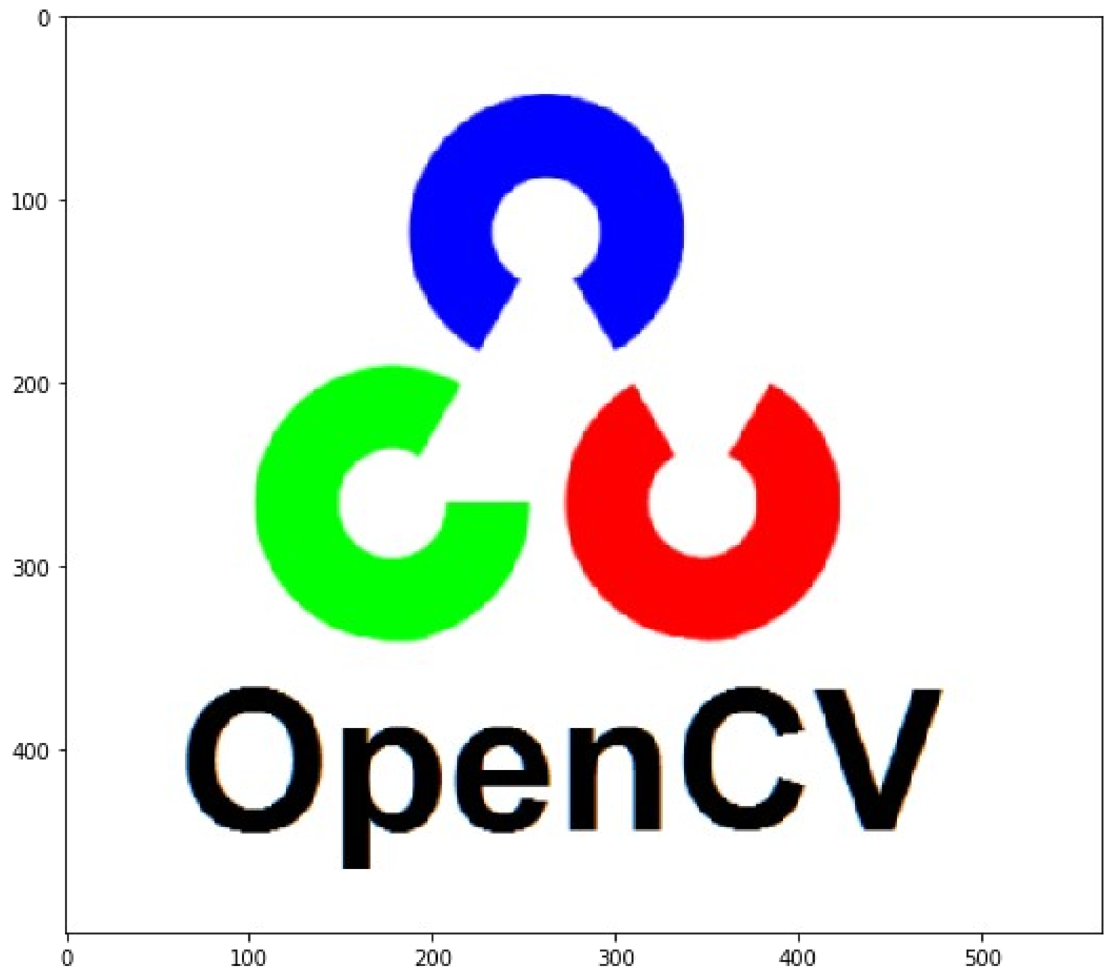
```
img = cv2.imread('imgs/opencv_logo1.png', 1)
# res = cv2.resize(img,None,fx=2.0, fy=2.0, interpolation =
cv2.INTER_CUBIC)
# #OR
height, width = img.shape[:2]
res = cv2.resize(img,(2*width, 2*height), interpolation =
cv2.INTER_CUBIC)

######################################################################
##########
# TO DO: Check the size of 'img' and 'res'?
######################################################################
##########
print(f'img size: {img.size}, img dimensions: {img.shape}\nres size:
{res.size}, res dimensions: {res.shape}')
######################################################################
##########
#                                  END OF YOUR CODE
#
######################################################################
##########

######################################################################
##########
# TO DO: Resize 'img' so as to the smaller side is 500, while keeping
image
# ration unchanged.
######################################################################
##########
scale = 500/min(img.shape[:2])
```

```
img2 = cv2.resize(img, None, fx=scale, fy=scale,
interpolation=cv2.INTER_LINEAR)

print(f'New img dimensions: {img2.shape}')
print(f'Ratio for original img: {img.shape[0]/img.shape[1]}\nRatio of
Modified img: {img2.shape[0]/img2.shape[1]}')

plt.figure(), plt.imshow(img)
plt.figure(), plt.imshow(img2)
######################################################################
##########
#                                  END OF YOUR CODE
#
######################################################################
##########
```

```
img size: 485352, img dimensions: (378, 428, 3)
res size: 1941408, res dimensions: (756, 856, 3)
New img dimensions: (500, 566, 3)
Ratio for original img: 0.883177570093458
Ratio of Modified img: 0.8833922261484098
```

```
(<Figure size 720x576 with 1 Axes>,
 <matplotlib.image.AxesImage at 0x2c83200adc0>)
```


OpenCV
0
50
100
150
200
250
300
350
400



Translation

Translation is the shifting of object's location. If you know the shift in (x, y) direction, let it be (t_x, t_y), you can create the transformation matrix M as follows:

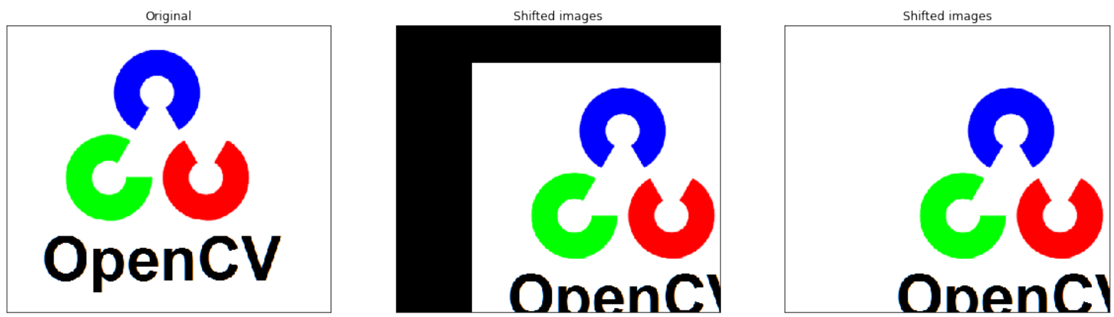
$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

You can take make it into a Numpy array of type **np.float32** and pass it into cv2.warpAffine() function.

```
img = cv2.imread('imgs/opencv_logo1.png', 1)
rows,cols,_ = img.shape
M = np.float32([[1,0,100],[0,1,50]]) # Shift right by 100 and down by
50
dst = cv2.warpAffine(img,M,(cols,rows))

######################################################################
##########
# TO DO: Observed that the bottom right of 'dst' image is lost.
```

```
Modifying the
# following codeline so as to the 'res' image is fully shown.
######################################################################
##########
# Get list of available flags
flags = [i for i in dir(cv2) if i.startswith('BORDER_')]
print(flags)

res = cv2.warpAffine(img,M,(cols,rows),
borderMode=cv2.BORDER_REPLICATE)

######################################################################
##########
#                                  END OF YOUR CODE
#
######################################################################
##########

plt.figure(figsize=(20,10))
plt.subplot(131),plt.imshow(img),
plt.title('Original'),plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(dst),
plt.title('Shifted images'),plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(res),
plt.title('Shifted images'),plt.xticks([]), plt.yticks([])
plt.show()
```

```
['BORDER_CONSTANT', 'BORDER_DEFAULT', 'BORDER_ISOLATED',
'BORDER_REFLECT', 'BORDER_REFLECT101', 'BORDER_REFLECT_101',
'BORDER_REPLICATE', 'BORDER_TRANSPARENT', 'BORDER_WRAP']
```



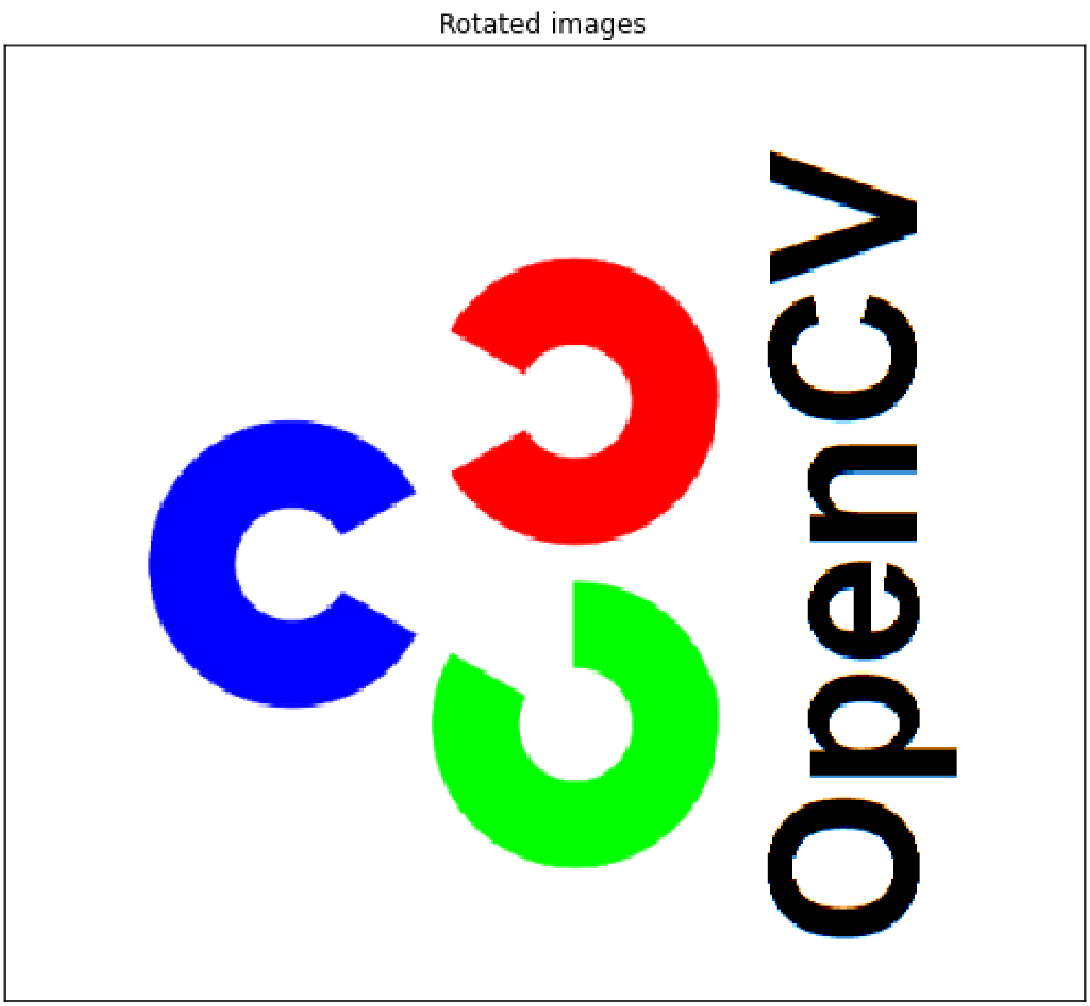
Rotation

Calculates an affine matrix of 2D rotation using cv2.getRotationMatrix2D().

- 1st argument: center
- 2nd argument: angle (in degree)
- 3rd argument: scale

```
img = cv2.imread('imgs/opencv_logo1.png', 1)
H,W,_ = img.shape
```

```
########################################################################
##########
# TO DO: Run the code to observe the output image.
# Modifying the code below so as to the 'dst' image has no black
padding.
########################################################################
##########
M = cv2.getRotationMatrix2D((W/2,H/2),90,1)
dst = cv2.warpAffine(img,M,(W,H), borderMode=cv2.BORDER_WRAP)
########################################################################
##########
#                                  END OF YOUR CODE
#
########################################################################
##########

plt.imshow(dst),
plt.title('Rotated images'),plt.xticks([]), plt.yticks([])
plt.show()
```

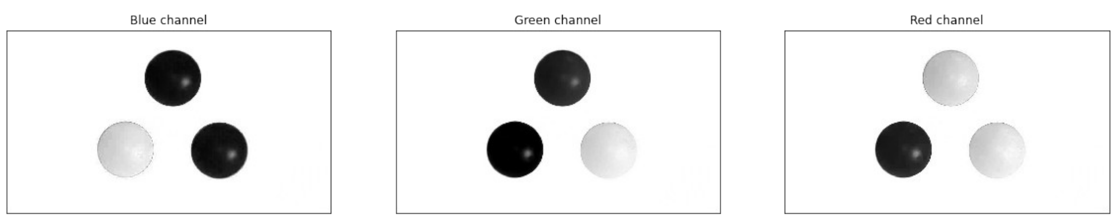


Changing color space - Grayscale

Grayscale values is converted from RGB values by a weighted sum of the R, G, and B components:

$$0.2989 \times R + 0.5870 \times G + 0.1140 \times B$$

```
# Split channels
img = cv2.imread('imgs/balls.jpg', 1)

plt.figure(figsize=(20,10))
plt.subplot(131),plt.imshow(img[:,:,0], cmap='gray'),
plt.title('Blue channel'),plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(img[:,:,1], cmap='gray'),
plt.title('Green channel'),plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(img[:,:,2], cmap='gray'),
plt.title('Red channel'),plt.xticks([]), plt.yticks([])
plt.show()
```

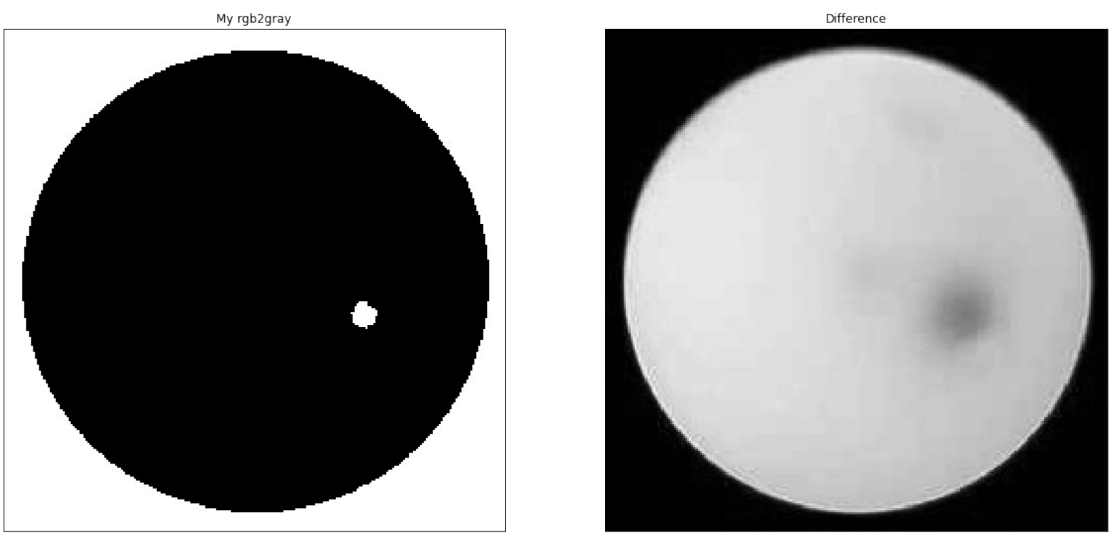



```
def rgb2gray(img):
    """
    A implementation of the method that converts BGR image to 
grayscale image of
    uint8 data type.
    """

######################################################################
##########
    # TO DO: Implement the method to convert BGR image to Grayscale 
image.         #
    # Hint: Remember to round and convert the values to nearest uint8 
values.      #

######################################################################
##########

    out = np.dot(img[...,:3], [0.2989, 0.5870, 0.1140])
    out = cv2.convertScaleAbs(out, alpha=(255.0/65535.0))

######################################################################
##########
    #                                  END OF YOUR CODE                
#
```

```
######################################################################
##########

    return out
```

Run the following code section to compare your implementation of the `rgb2gray` function with OpenCV built-in function cv2.cvtColor.

```
img = cv2.imread('imgs/ball_red.jpg', 1)
img_gray1 = rgb2gray(img)
img_gray2 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

plt.figure(figsize=(20,10))
plt.subplot(121),plt.imshow(img_gray1, cmap='gray'),
plt.title('My rgb2gray'),plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(img_gray1 - img_gray2, cmap='gray'),
plt.title('Difference'),plt.xticks([]), plt.yticks([])
plt.show()

# Check your output: count
print('Testing rgb2gray')
print('Number of difference pixel is %d' % np.count_nonzero(img_gray1
- img_gray2))
```




```
Testing rgb2gray
Number of difference pixel is 57121
```

Question: Does your implementation of `rgb2gray` function give the result that is exactly the same as OpenCV built-in function? Why?

Your answer: No. The cv2 function gives a different output. OpenCV uses nearest integer rounding https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html

Changing color space - Detect object by color.

By converting BGR image to HSV, we can use this to extract a colored object. In HSV, it is more easier to represent a color than RGB color-space. In this exercise, we will try to extract blue, red, and yellow colored objects. So here is the method:

- Take each frame of the video
- Convert from BGR to HSV color-space
- We threshold the HSV image for a range of blue color
- Now extract the blue object alone, we can do whatever on that image we want.

```
# Get list of available flags
flags = [i for i in dir(cv2) if i.startswith('COLOR_')]
print(flags)
```

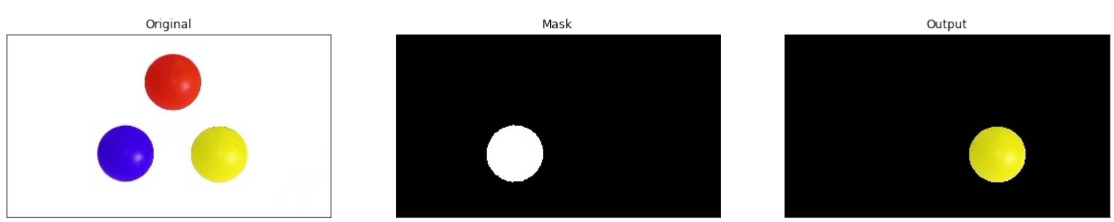
```
['COLOR_BAYER_BG2BGR', 'COLOR_BAYER_BG2BGRA', 'COLOR_BAYER_BG2BGR_EA',
'COLOR_BAYER_BG2BGR_VNG', 'COLOR_BAYER_BG2GRAY', 'COLOR_BAYER_BG2RGB',
'COLOR_BAYER_BG2RGBA', 'COLOR_BAYER_BG2RGB_EA',
'COLOR_BAYER_BG2RGB_VNG', 'COLOR_BAYER_BGGR2BGR',
'COLOR_BAYER_BGGR2BGRA', 'COLOR_BAYER_BGGR2BGR_EA',
'COLOR_BAYER_BGGR2BGR_VNG', 'COLOR_BAYER_BGGR2GRAY',
'COLOR_BAYER_BGGR2RGB', 'COLOR_BAYER_BGGR2RGBA',
'COLOR_BAYER_BGGR2RGB_EA', 'COLOR_BAYER_BGGR2RGB_VNG',
'COLOR_BAYER_GB2BGR', 'COLOR_BAYER_GB2BGRA', 'COLOR_BAYER_GB2BGR_EA',
'COLOR_BAYER_GB2BGR_VNG', 'COLOR_BAYER_GB2GRAY', 'COLOR_BAYER_GB2RGB',
'COLOR_BAYER_GB2RGBA', 'COLOR_BAYER_GB2RGB_EA',
'COLOR_BAYER_GB2RGB_VNG', 'COLOR_BAYER_GBRG2BGR',
'COLOR_BAYER_GBRG2BGRA', 'COLOR_BAYER_GBRG2BGR_EA',
'COLOR_BAYER_GBRG2BGR_VNG', 'COLOR_BAYER_GBRG2GRAY',
'COLOR_BAYER_GBRG2RGB', 'COLOR_BAYER_GBRG2RGBA',
'COLOR_BAYER_GBRG2RGB_EA', 'COLOR_BAYER_GBRG2RGB_VNG',
'COLOR_BAYER_GR2BGR', 'COLOR_BAYER_GR2BGRA', 'COLOR_BAYER_GR2BGR_EA',
'COLOR_BAYER_GR2BGR_VNG', 'COLOR_BAYER_GR2GRAY', 'COLOR_BAYER_GR2RGB',
'COLOR_BAYER_GR2RGBA', 'COLOR_BAYER_GR2RGB_EA',
'COLOR_BAYER_GR2RGB_VNG', 'COLOR_BAYER_GRBG2BGR',
'COLOR_BAYER_GRBG2BGRA', 'COLOR_BAYER_GRBG2BGR_EA',
'COLOR_BAYER_GRBG2BGR_VNG', 'COLOR_BAYER_GRBG2GRAY',
'COLOR_BAYER_GRBG2RGB', 'COLOR_BAYER_GRBG2RGBA',
'COLOR_BAYER_GRBG2RGB_EA', 'COLOR_BAYER_GRBG2RGB_VNG',
'COLOR_BAYER_RG2BGR', 'COLOR_BAYER_RG2BGRA', 'COLOR_BAYER_RG2BGR_EA',
'COLOR_BAYER_RG2BGR_VNG', 'COLOR_BAYER_RG2GRAY', 'COLOR_BAYER_RG2RGB',
'COLOR_BAYER_RG2RGBA', 'COLOR_BAYER_RG2RGB_EA',
'COLOR_BAYER_RG2RGB_VNG', 'COLOR_BAYER_RGGB2BGR',
'COLOR_BAYER_RGGB2BGRA', 'COLOR_BAYER_RGGB2BGR_EA',
'COLOR_BAYER_RGGB2BGR_VNG', 'COLOR_BAYER_RGGB2GRAY',
'COLOR_BAYER_RGGB2RGB', 'COLOR_BAYER_RGGB2RGBA',
'COLOR_BAYER_RGGB2RGB_EA', 'COLOR_BAYER_RGGB2RGB_VNG',
```

```
'COLOR_BGR2BGR555', 'COLOR_BGR2BGR565', 'COLOR_BGR2BGRA',
'COLOR_BGR2GRAY', 'COLOR_BGR2HLS', 'COLOR_BGR2HLS_FULL',
'COLOR_BGR2HSV', 'COLOR_BGR2HSV_FULL', 'COLOR_BGR2LAB',
'COLOR_BGR2LUV', 'COLOR_BGR2Lab', 'COLOR_BGR2Luv', 'COLOR_BGR2RGB',
'COLOR_BGR2RGBA', 'COLOR_BGR2XYZ', 'COLOR_BGR2YCR_CB',
'COLOR_BGR2YCrCb', 'COLOR_BGR2YUV', 'COLOR_BGR2YUV_I420',
'COLOR_BGR2YUV_IYUV', 'COLOR_BGR2YUV_YV12', 'COLOR_BGR5552BGR',
'COLOR_BGR5552BGRA', 'COLOR_BGR5552GRAY', 'COLOR_BGR5552RGB',
'COLOR_BGR5552RGBA', 'COLOR_BGR5652BGR', 'COLOR_BGR5652BGRA',
'COLOR_BGR5652GRAY', 'COLOR_BGR5652RGB', 'COLOR_BGR5652RGBA',
'COLOR_BGRA2BGR', 'COLOR_BGRA2BGR555', 'COLOR_BGRA2BGR565',
'COLOR_BGRA2GRAY', 'COLOR_BGRA2RGB', 'COLOR_BGRA2RGBA',
'COLOR_BGRA2YUV_I420', 'COLOR_BGRA2YUV_IYUV', 'COLOR_BGRA2YUV_YV12',
'COLOR_BayerBG2BGR', 'COLOR_BayerBG2BGRA', 'COLOR_BayerBG2BGR_EA',
'COLOR_BayerBG2BGR_VNG', 'COLOR_BayerBG2GRAY', 'COLOR_BayerBG2RGB',
'COLOR_BayerBG2RGBA', 'COLOR_BayerBG2RGB_EA', 'COLOR_BayerBG2RGB_VNG',
'COLOR_BayerBGGR2BGR', 'COLOR_BayerBGGR2BGRA',
'COLOR_BayerBGGR2BGR_EA', 'COLOR_BayerBGGR2BGR_VNG',
'COLOR_BayerBGGR2GRAY', 'COLOR_BayerBGGR2RGB', 'COLOR_BayerBGGR2RGBA',
'COLOR_BayerBGGR2RGB_EA', 'COLOR_BayerBGGR2RGB_VNG',
'COLOR_BayerGB2BGR', 'COLOR_BayerGB2BGRA', 'COLOR_BayerGB2BGR_EA',
'COLOR_BayerGB2BGR_VNG', 'COLOR_BayerGB2GRAY', 'COLOR_BayerGB2RGB',
'COLOR_BayerGB2RGBA', 'COLOR_BayerGB2RGB_EA', 'COLOR_BayerGB2RGB_VNG',
'COLOR_BayerGBRG2BGR', 'COLOR_BayerGBRG2BGRA',
'COLOR_BayerGBRG2BGR_EA', 'COLOR_BayerGBRG2BGR_VNG',
'COLOR_BayerGBRG2GRAY', 'COLOR_BayerGBRG2RGB', 'COLOR_BayerGBRG2RGBA',
'COLOR_BayerGBRG2RGB_EA', 'COLOR_BayerGBRG2RGB_VNG',
'COLOR_BayerGR2BGR', 'COLOR_BayerGR2BGRA', 'COLOR_BayerGR2BGR_EA',
'COLOR_BayerGR2BGR_VNG', 'COLOR_BayerGR2GRAY', 'COLOR_BayerGR2RGB',
'COLOR_BayerGR2RGBA', 'COLOR_BayerGR2RGB_EA', 'COLOR_BayerGR2RGB_VNG',
'COLOR_BayerGRBG2BGR', 'COLOR_BayerGRBG2BGRA',
'COLOR_BayerGRBG2BGR_EA', 'COLOR_BayerGRBG2BGR_VNG',
'COLOR_BayerGRBG2GRAY', 'COLOR_BayerGRBG2RGB', 'COLOR_BayerGRBG2RGBA',
'COLOR_BayerGRBG2RGB_EA', 'COLOR_BayerGRBG2RGB_VNG',
'COLOR_BayerRG2BGR', 'COLOR_BayerRG2BGRA', 'COLOR_BayerRG2BGR_EA',
'COLOR_BayerRG2BGR_VNG', 'COLOR_BayerRG2GRAY', 'COLOR_BayerRG2RGB',
'COLOR_BayerRG2RGBA', 'COLOR_BayerRG2RGB_EA', 'COLOR_BayerRG2RGB_VNG',
'COLOR_BayerRGGB2BGR', 'COLOR_BayerRGGB2BGRA',
'COLOR_BayerRGGB2BGR_EA', 'COLOR_BayerRGGB2BGR_VNG',
'COLOR_BayerRGGB2GRAY', 'COLOR_BayerRGGB2RGB', 'COLOR_BayerRGGB2RGBA',
'COLOR_BayerRGGB2RGB_EA', 'COLOR_BayerRGGB2RGB_VNG',
'COLOR_COLORCVT_MAX', 'COLOR_GRAY2BGR', 'COLOR_GRAY2BGR555',
'COLOR_GRAY2BGR565', 'COLOR_GRAY2BGRA', 'COLOR_GRAY2RGB',
'COLOR_GRAY2RGBA', 'COLOR_HLS2BGR', 'COLOR_HLS2BGR_FULL',
'COLOR_HLS2RGB', 'COLOR_HLS2RGB_FULL', 'COLOR_HSV2BGR',
'COLOR_HSV2BGR_FULL', 'COLOR_HSV2RGB', 'COLOR_HSV2RGB_FULL',
'COLOR_LAB2BGR', 'COLOR_LAB2LBGR', 'COLOR_LAB2LRGB', 'COLOR_LAB2RGB',
'COLOR_LBGR2LAB', 'COLOR_LBGR2LUV', 'COLOR_LBGR2Lab',
'COLOR_LBGR2Luv', 'COLOR_LRGB2LAB', 'COLOR_LRGB2LUV',
'COLOR_LRGB2Lab', 'COLOR_LRGB2Luv', 'COLOR_LUV2BGR', 'COLOR_LUV2LBGR',
```

```
'COLOR_LUV2LRGB', 'COLOR_LUV2RGB', 'COLOR_Lab2BGR', 'COLOR_Lab2LBGR',
'COLOR_Lab2LRGB', 'COLOR_Lab2RGB', 'COLOR_Luv2BGR', 'COLOR_Luv2LBGR',
'COLOR_Luv2LRGB', 'COLOR_Luv2RGB', 'COLOR_M_RGBA2RGBA',
'COLOR_RGB2BGR', 'COLOR_RGB2BGR555', 'COLOR_RGB2BGR565',
'COLOR_RGB2BGRA', 'COLOR_RGB2GRAY', 'COLOR_RGB2HLS',
'COLOR_RGB2HLS_FULL', 'COLOR_RGB2HSV', 'COLOR_RGB2HSV_FULL',
'COLOR_RGB2LAB', 'COLOR_RGB2LUV', 'COLOR_RGB2Lab', 'COLOR_RGB2Luv',
'COLOR_RGB2RGBA', 'COLOR_RGB2XYZ', 'COLOR_RGB2YCR_CB',
'COLOR_RGB2YCrCb', 'COLOR_RGB2YUV', 'COLOR_RGB2YUV_I420',
'COLOR_RGB2YUV_IYUV', 'COLOR_RGB2YUV_YV12', 'COLOR_RGBA2BGR',
'COLOR_RGBA2BGR555', 'COLOR_RGBA2BGR565', 'COLOR_RGBA2BGRA',
'COLOR_RGBA2GRAY', 'COLOR_RGBA2M_RGBA', 'COLOR_RGBA2RGB',
'COLOR_RGBA2YUV_I420', 'COLOR_RGBA2YUV_IYUV', 'COLOR_RGBA2YUV_YV12',
'COLOR_RGBA2mRGBA', 'COLOR_XYZ2BGR', 'COLOR_XYZ2RGB',
'COLOR_YCR_CB2BGR', 'COLOR_YCR_CB2RGB', 'COLOR_YCrCb2BGR',
'COLOR_YCrCb2RGB', 'COLOR_YUV2BGR', 'COLOR_YUV2BGRA_I420',
'COLOR_YUV2BGRA_IYUV', 'COLOR_YUV2BGRA_NV12', 'COLOR_YUV2BGRA_NV21',
'COLOR_YUV2BGRA_UYNV', 'COLOR_YUV2BGRA_UYVY', 'COLOR_YUV2BGRA_Y422',
'COLOR_YUV2BGRA_YUNV', 'COLOR_YUV2BGRA_YUY2', 'COLOR_YUV2BGRA_YUYV',
'COLOR_YUV2BGRA_YV12', 'COLOR_YUV2BGRA_YVYU', 'COLOR_YUV2BGR_I420',
'COLOR_YUV2BGR_IYUV', 'COLOR_YUV2BGR_NV12', 'COLOR_YUV2BGR_NV21',
'COLOR_YUV2BGR_UYNV', 'COLOR_YUV2BGR_UYVY', 'COLOR_YUV2BGR_Y422',
'COLOR_YUV2BGR_YUNV', 'COLOR_YUV2BGR_YUY2', 'COLOR_YUV2BGR_YUYV',
'COLOR_YUV2BGR_YV12', 'COLOR_YUV2BGR_YVYU', 'COLOR_YUV2GRAY_420',
'COLOR_YUV2GRAY_I420', 'COLOR_YUV2GRAY_IYUV', 'COLOR_YUV2GRAY_NV12',
'COLOR_YUV2GRAY_NV21', 'COLOR_YUV2GRAY_UYNV', 'COLOR_YUV2GRAY_UYVY',
'COLOR_YUV2GRAY_Y422', 'COLOR_YUV2GRAY_YUNV', 'COLOR_YUV2GRAY_YUY2',
'COLOR_YUV2GRAY_YUYV', 'COLOR_YUV2GRAY_YV12', 'COLOR_YUV2GRAY_YVYU',
'COLOR_YUV2RGB', 'COLOR_YUV2RGBA_I420', 'COLOR_YUV2RGBA_IYUV',
'COLOR_YUV2RGBA_NV12', 'COLOR_YUV2RGBA_NV21', 'COLOR_YUV2RGBA_UYNV',
'COLOR_YUV2RGBA_UYVY', 'COLOR_YUV2RGBA_Y422', 'COLOR_YUV2RGBA_YUNV',
'COLOR_YUV2RGBA_YUY2', 'COLOR_YUV2RGBA_YUYV', 'COLOR_YUV2RGBA_YV12',
'COLOR_YUV2RGBA_YVYU', 'COLOR_YUV2RGB_I420', 'COLOR_YUV2RGB_IYUV',
'COLOR_YUV2RGB_NV12', 'COLOR_YUV2RGB_NV21', 'COLOR_YUV2RGB_UYNV',
'COLOR_YUV2RGB_UYVY', 'COLOR_YUV2RGB_Y422', 'COLOR_YUV2RGB_YUNV',
'COLOR_YUV2RGB_YUY2', 'COLOR_YUV2RGB_YUYV', 'COLOR_YUV2RGB_YV12',
'COLOR_YUV2RGB_YVYU', 'COLOR_YUV420P2BGR', 'COLOR_YUV420P2BGRA',
'COLOR_YUV420P2GRAY', 'COLOR_YUV420P2RGB', 'COLOR_YUV420P2RGBA',
'COLOR_YUV420SP2BGR', 'COLOR_YUV420SP2BGRA', 'COLOR_YUV420SP2GRAY',
'COLOR_YUV420SP2RGB', 'COLOR_YUV420SP2RGBA', 'COLOR_YUV420p2BGR',
'COLOR_YUV420p2BGRA', 'COLOR_YUV420p2GRAY', 'COLOR_YUV420p2RGB',
'COLOR_YUV420p2RGBA', 'COLOR_YUV420sp2BGR', 'COLOR_YUV420sp2BGRA',
'COLOR_YUV420sp2GRAY', 'COLOR_YUV420sp2RGB', 'COLOR_YUV420sp2RGBA',
'COLOR_mRGBA2RGBA']
```

```python
frame = cv2.imread('imgs/balls.jpg', 1)

# Convert BGR to RGB, now you will see the color of 'frame' image
# is displayed properly.
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
```

```python
# Convert BGR to HSV
hsv = cv2.cvtColor(frame, cv2.COLOR_RGB2HSV)

# define range of blue color in HSV
lower_blue = np.array([110,50,50])
upper_blue = np.array([130,255,255])

# Threshold the HSV image to get only blue colors
mask = cv2.inRange(hsv, lower_blue, upper_blue)

# Bitwise-AND mask and original image
res = cv2.bitwise_and(frame,frame, mask= mask)

######################################################################
##########
# TO DO: Implement masks for red and yellow balls.
######################################################################
##########
 # yellow mask
yellow_lower = np.array([20, 100, 100])
yellow_upper = np.array([60, 255, 255])
mask_yellow = cv2.inRange(hsv, yellow_lower, yellow_upper)

# Bitwise-AND mask and original image
res = cv2.bitwise_and(frame,frame, mask=mask_yellow)
######################################################################
##########
#                                  END OF YOUR CODE
#
######################################################################
##########

plt.figure(figsize=(20,10))
plt.subplot(131),plt.imshow(frame),
plt.title('Original'),plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(mask, cmap='gray'),
plt.title('Mask'),plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(res),
plt.title('Output'),plt.xticks([]), plt.yticks([])
plt.show()
```



2D Convolution (Image Filtering)

OpenCV provides a function, cv2.filter2D, to convolve a kernel with an image.

```
def convolution_naive(x, F, conv_param):
    """
    A naive implementation of a convolutional filter.

    The input consists of a gray scale image x (1 channel) with height
H and width
    W. We convolve each input with filter F, which has height HH and
width HH.

    Input:
    - x: Input data of shape (H, W)
    - F: Filter weights of shape (HH, WW)
    - conv_param: A dictionary with the following keys:
      - 'stride': The number of pixels between adjacent receptive
fields in the
        horizontal and vertical directions.
      - 'pad': The number of pixels that will be used to zero-pad the
input.

    Return:
    - out: Output data, of shape (H', W') where H' and W' are given by
      H' = 1 + (H + 2 * pad - HH) / stride
      W' = 1 + (W + 2 * pad - WW) / stride
    """

    stride = conv_param['stride']
    pad = conv_param['pad']
    H, W = x.shape
    HH, WW = F.shape
    H_prime = int(1 + (H + 2 * pad - HH) / stride)
    W_prime = int(1 + (W + 2 * pad - WW) / stride)
    x_pad = np.lib.pad(x, ((pad, pad), (pad, pad)),\
                            'constant', constant_values=(0))
    out = np.zeros((H_prime, W_prime), dtype=x.dtype)
    print(x_pad.shape)

######################################################################
#######
    # TODO: Implement the convolutional forward pass.
#
    # Hint: Using 2 nested for-loop to calculate each pixel of the
output image.#

######################################################################
#######
    for y_index in range(0, H_prime,stride):
```

```
        for x_index in range(0, W_prime,stride):
            field = x_pad[x_index: x_index + WW, y_index: y_index +
HH] # Move window over correct area in image
            out[x_index, y_index] = np.sum(np.multiply(F, field))


######################################################################
#######
    #                             END OF YOUR CODE
#

######################################################################
#######
    return out
```

Run the following code section to test your implementation of the `convolution_naive` function

```
x_shape = (5, 5)
F_shape = (3, 3)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
F = np.linspace(-0.2, 0.3, num=np.prod(F_shape)).reshape(F_shape)
conv_param = {'stride': 1, 'pad': 1}

out = convolution_naive(x, F, conv_param)
correct_out = np.array( [[ 0.0075,     0.030625,   0.0521875,
0.07375,    0.0475   ],
                         [ 0.114375,   0.1725,     0.18375,    0.195,
0.10875  ],
                         [ 0.1753125,  0.22875,    0.24,
0.25125,    0.1228125],
                         [ 0.23625,    0.285,      0.29625,    0.3075,
0.136875 ],
                         [ 0.0075,    -0.05375,   -0.0603125, -
0.066875,  -0.1025   ]])
print(correct_out.shape, out.shape)
print(out)

# Compare your output to ours; difference should be very small
print('Testing convolution_naive')
print('difference: ', rel_error(out, correct_out))
```

```
(7, 7)
(5, 5) (5, 5)
[[ 0.0075     0.030625   0.0521875  0.07375    0.0475   ]
 [ 0.114375   0.1725     0.18375    0.195      0.10875  ]
 [ 0.1753125  0.22875    0.24       0.25125    0.1228125]
 [ 0.23625    0.285      0.29625    0.3075     0.136875 ]
 [ 0.0075    -0.05375   -0.0603125 -0.066875  -0.1025   ]]
```

```
Testing convolution_naive
difference:  0.0
```

```
# List of available BORDER effect
flags = [i for i in dir(cv2) if i.startswith('BORDER_')]
print(flags)
```

```
['BORDER_CONSTANT', 'BORDER_DEFAULT', 'BORDER_ISOLATED',
'BORDER_REFLECT', 'BORDER_REFLECT101', 'BORDER_REFLECT_101',
'BORDER_REPLICATE', 'BORDER_TRANSPARENT', 'BORDER_WRAP']
```

Averaging filter

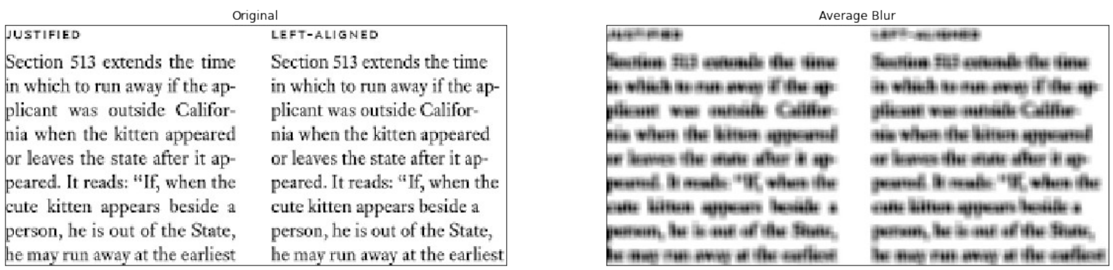
This is done by convolving image with a normalized box filter. A 5×5 normalized box filter would look like below:

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

```
# Convert image data type from uint8 to float32.
img = cv2.imread('imgs/text.png', 1).astype(np.float32)
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

kernel = np.zeros((5,5), np.float32)
######################################################################
##########
# TODO: Create a 5x5 kernel as K shown above.
#
######################################################################
##########
kernel = np.ones((5, 5),  np.float32) * (0.04)
######################################################################
##########
#                                  END OF YOUR CODE
#
######################################################################
##########
blur_2dfilter = cv2.filter2D(img,-1,kernel)

# The above codes can be replaced by the following code line.
blur = cv2.blur(img,(5,5))

# Check your output; difference should be around 4e-3
print('Testing convolution_naive')
print('difference: ', rel_error(blur_2dfilter, blur))
```

```
# Visualize the output image
plt.figure(figsize=(20,10))
plt.subplot(121),plt.imshow(img, cmap='gray'),
plt.title('Original'),plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(blur, cmap='gray'),
plt.title('Average Blur'),plt.xticks([]), plt.yticks([])
plt.show()
```

```
Testing convolution_naive
difference:  0.0035056125
```



Gaussian Blurring

Here is the 1D Gaussian distribution:

$$G(x)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{x^2}{\sigma^2}\right)$$

1D Gaussian

Similarly, we have 2D Gaussian distribution.

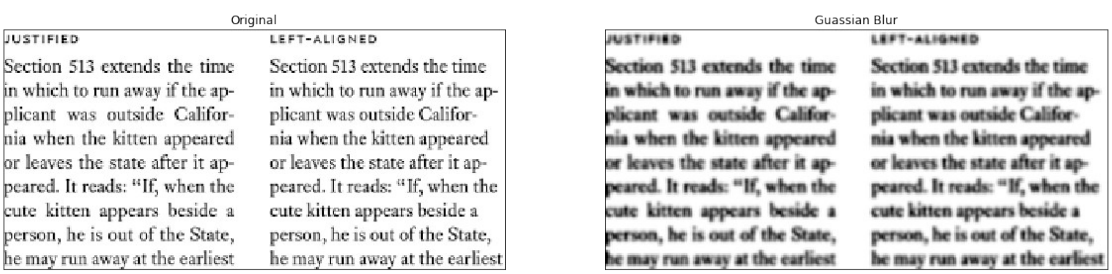
$$G(x,y)=\frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{\sigma^2}\right)$$

The nearest neighboring pixels have the most influence. 2D Gaussian

```
img = cv2.imread('imgs/text.png', 1).astype(np.float32)/255.0
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

gaussian_kernel_XY = np.zeros((5,5), np.float32)
######################################################################
##########
# TODO: Create a 5x5 kernel, 'gaussian_kernel_XY', which approximates
the
# Gaussian function with sigma=1.
# Hint: + You should NOT munually create the kernel.
#       + Use the 'cv2.getGaussianKernel' function to create 1D
Guassian kernel.
#       + Use the associative property of convolution to create 2D
Gaussian. A
# useful reference:
```

```
https://blogs.mathworks.com/steve/2006/10/04/separable-convolution/
######################################################################
##########
k = cv2.getGaussianKernel(5, 1)
gaussian_kernel_XY = k * k.T

######################################################################
##########
#                                  END OF YOUR CODE
#
######################################################################
##########
blur_2dfilter = cv2.filter2D(img,-1,gaussian_kernel_XY)

# The above codes can be replaced by the following code line.
blur = cv2.GaussianBlur(img,(5,5),1)

# Check your output; difference should be around 4e-3
print('Testing convolution_naive')
print('difference: ', rel_error(blur_2dfilter, blur))

# Visualize the output image
plt.figure(figsize=(20,10))
plt.subplot(121),plt.imshow(img, cmap='gray'),
plt.title('Original'),plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(blur, cmap='gray'),
plt.title('Guassian Blur'),plt.xticks([]), plt.yticks([])
plt.show()
```

```
Testing convolution_naive
difference:  0.0042602094
```



QUESTION: Provide your comments on the outputs of *a average filter* and *a Gaussian filter*? Which one is more preferable?

Your answer: The average filter produces a more blurred image than the Gaussian filter. The words in the image produced by the average filter are not legible at all since it blurs the entire image equally. The gaussian blur is able to retain characteristics in the original image, and words are still somewhat legible since the filter gives a greater weight to pixels closer to the centre of the kernel, such that the central pixels are less blurred. The average filter is good at reducing random noise in a spatial domain (low-pass), but often passes many high-frequency features while removing some low-frequency features in the image low frequencies (slow roll-off and poor stopband attenuation). On the other hand, gaussian filter is better at separating out frequencies in the image but much more computationally intensive. In the case of the image provided, the gaussian filter would be preferable since the image has a large contrast, and the gaussian filter would be able to amplify the word clusters better.

Median Filter

Example:

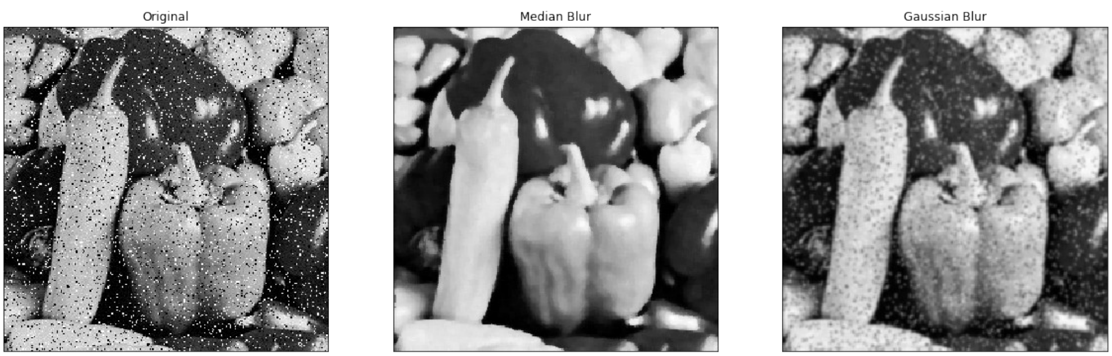
- **Odd** number of elements: $X=[2,5,1,0,9] \rightarrow X_{sorted}=[0,1,2,5,9] \Rightarrow \text{median}=2$
- **Even** number of elements:
  - Option 1: $X=[5,1,0,9] \rightarrow X_{sorted}=[0,1,5,9] \Rightarrow \text{median}=1$
  - Option 2: $X=[5,1,0,9] \rightarrow X_{sorted}=[0,1,5,9] \Rightarrow \text{median}=(1+5)/2=3$

```
# Implement a function to find median value with `option 1`.
def findMedian(x):
    out = 0

######################################################################
#######
    # TODO: Implement the function to find median value of array x.
#
    # NOTE: You should see that the `median' numpy built-in function
is based    #
    # on option 2.

######################################################################
#######
    sorted = np.sort(x, axis=None)
    if len(sorted)%2 == 0:
        # even
        out = sorted[len(sorted)//2 - 1]

    else:
        #odd
        index = len(sorted)//2 + 1
        out = sorted[index]

######################################################################
#######
    #                              END OF YOUR CODE
#

######################################################################
#######
    return out
```

```
print ('Numpy median: ', np.median([[5,1],[0,9]]))
print ('Numpy median: ', np.median([2,5,1,0,9]))
print ('findMedian: ', findMedian([[5,1],[0,9]]))
print ('findMedian: ', findMedian([2,5,1,0,9]))
```

```
Numpy median:  3.0
Numpy median:  2.0
findMedian:  1
findMedian:  5
```

```
img = cv2.imread('imgs/SaltAndPepperNoise.jpg', 0)
median = cv2.medianBlur(img,5)
gau_blur = cv2.GaussianBlur(img,(5,5),1)

plt.figure(figsize=(20,10))
plt.subplot(131),plt.imshow(img, 'gray')
plt.title('Original'),plt.xticks([]),plt.yticks([])
plt.subplot(132),plt.imshow(median, 'gray')
plt.title('Median Blur'),plt.xticks([]),plt.yticks([])
plt.subplot(133),plt.imshow(gau_blur, 'gray')
plt.title('Gaussian Blur'),plt.xticks([]),plt.yticks([])
plt.show()
```



QUESTION: Provide your comments on the effectiveness of *a median filter* and *a Gaussian filter* for the example above? Explain why?

Your answer: The median filter is more effective than the gaussian filter here. The gaussian filter is a type of linear filter which averages over the window, with central pixels having a higher weight. It does not preserve image edges. The median filter is a non-linear filter which removes noise while preserving the edges of an image. The median filter removes noise by replacing pixel values with the median pixel value present in the window. In the original image, the pixel values of noise are either much lighter or much darker than the rest of the pixels, and will be replaced by the median pixel value of the image content instead. On the other hand, the gaussian filter does not remove the noisy pixels, and simply produces a image with lower contrast than the original.

```
def myMedianBlur(img, size):
    """
    A implementation of median blur filter.
```

```
    """
    out = img.copy()
    W,H = img.shape[0],img.shape[1]
    s = (size - 1)/2

#####################################################################
#######
    # TODO: Implement the median blur.
#
    # NOTE: Your implementation is NOT necessary to provide the
identical       #
    # output as OpenCV built-in function. However, it should be
visually very   #
    # similar.
#

#####################################################################
#######
    s = int(s)
    for y_index in range(s, H-s, size):
        for x_index in range(s, W-s, size):
            med = findMedian(img[x_index-s: x_index + s, y_index-s:
y_index + s])
            out[x_index-s: x_index + s, y_index-s: y_index + s] = med

#####################################################################
#######
    #                             END OF YOUR CODE
#

#####################################################################
#######
    return out

img = cv2.imread('imgs/SaltAndPepperNoise.jpg', 0)
mymedian = myMedianBlur(img,5)
median = cv2.medianBlur(img,5)

# Note that your implementation is NOT necessary to provide
# the identical output as OpenCV built-in function. However,
# it should visually very similar.
plt.figure(figsize=(16,8))
plt.subplot(121),plt.imshow(median, 'gray')
plt.title('Opencv Median Blur'),plt.xticks([]),plt.yticks([])
plt.subplot(122),plt.imshow(median, 'gray')
plt.title('My Median Blur'),plt.xticks([]),plt.yticks([])
plt.show()
```

Image gradient

For 1-D continuous function $f(x)$, the gradient is given as:

$$D_x[f(x)] = \frac{d}{dx} f(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}, \text{or} \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x}$$

For 1-D discrete function $f[n]$, the gradient becomes difference.

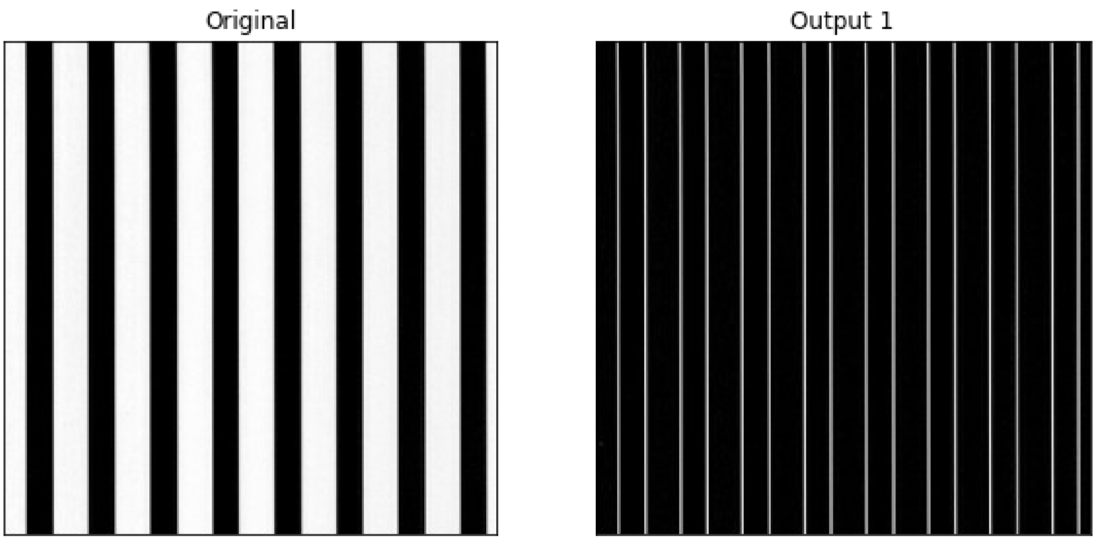
$$D_n[f[n]] = f[n+1] - f[n], \text{or} \frac{f[n+1] - f[n-1]}{2}$$

The kernel to find gradient in 1-D discrete function is $[1, 0, -1]$.

```
img = cv2.imread('imgs/banded_vertical.jpg', 0).astype(np.float32)

#####################################################################
#######
# TODO: Create a 3x3 kernel, Kx, to find the gradient in x-axis of an
image.#
#####################################################################
#######
Kx = np.tile([1, 0, -1], (3, 1))
print(Kx)
#####################################################################
#######
#                              END OF YOUR CODE
#
#####################################################################
#######
dstx = cv2.filter2D(img,-1, Kx)
```

```
plt.figure(figsize=(10,5))
plt.subplot(121),plt.imshow(img, cmap='gray')
plt.title('Original'),plt.xticks([]),plt.yticks([])
plt.subplot(122),plt.imshow(np.abs(dstx), cmap='gray')
plt.title('Output 1'),plt.xticks([]),plt.yticks([])
plt.show()
```

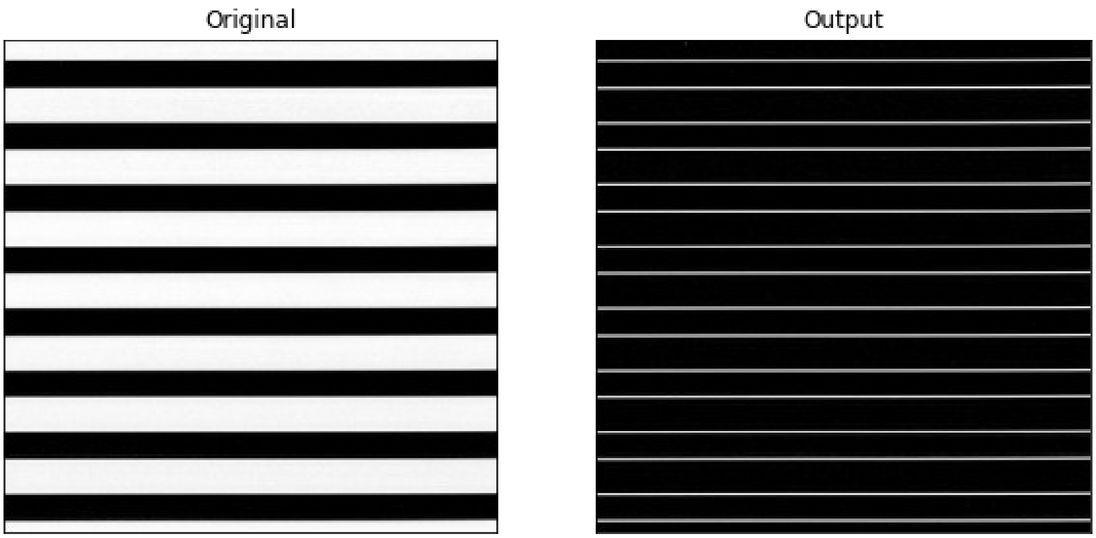
```
[[ 1  0 -1]
 [ 1  0 -1]
 [ 1  0 -1]]
```




```
img = cv2.imread('imgs/banded_horizontal.jpg', 0).astype(np.float32)

######################################################################
#######
# TODO: Create a 3x3 kernel, Ky, to find the gradient in y-axis of an
image.#
######################################################################
#######
Ky = np.tile(np.array([[1, 0, -1]]).T, (1, 3))
print(Ky)
######################################################################
#######
#                              END OF YOUR CODE
#
######################################################################
#######
dsty = cv2.filter2D(img,-1,Ky)


plt.figure(figsize=(10,5))
plt.subplot(121),plt.imshow(img, 'gray')
plt.title('Original'),plt.xticks([]),plt.yticks([])
```

```
plt.subplot(122),plt.imshow(np.abs(dsty), 'gray')
plt.title('Output'),plt.xticks([]),plt.yticks([])
plt.show()
```

```
[[ 1  1  1]
 [ 0  0  0]
 [-1 -1 -1]]
```



Question: What do the kernel Kx and Ky do in *image processing*?

Answer: Kx and Ky detects the change in pixel value as the kernel moves horizontally (for Kx) or vertically (for Ky). When the window passes over a part of the image with all black or all white pixels, there is no change in pixel value, the gradient is 0, and the output is black. When the kernel moves over the region containing black and white lines, there is a change in pixel value, gradient is non-zero and the output is white.

Two directions:

- Find the difference: in the two directions:

$$g_x[m,n] = f[m+1,n] - f[m-1,n]$$

$$g_y[m,n] = f[m,n+1] - f[m,n-1]$$

- Find the magnitude and direction of the gradient vector:

$$\| g[m,n] \| = \sqrt{g_x^2[m,n] + g_y^2[m,n]}$$

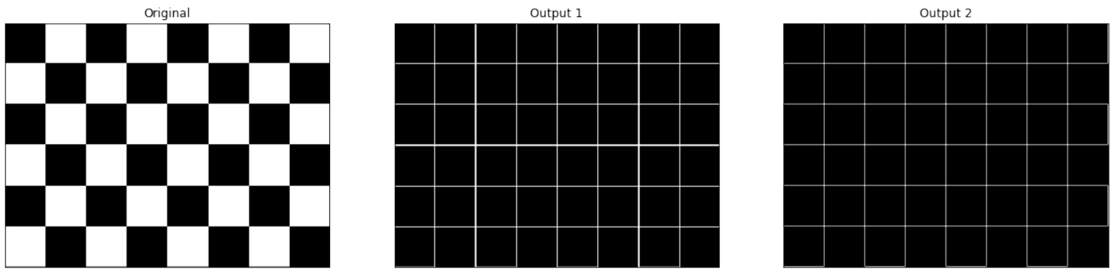
$$\measuredangle g[m,n] = \tan^{-1}\left(\frac{g_y[m,n]}{g_x[m,n]}\right)$$

```
img = cv2.imread('imgs/chequered.jpg', 0).astype(np.float32)

###############################################################################
```

```
# TODO: Using the theory provided above, compute the magnitude of 2
#
# direction image gradient.
#
#########################################################################
#######
kgx = np.array([[0,0,0],
              [-1,0,1],
              [0,0,0]], dtype=np.float32)
kgy = np.array([[0,-1,0],
              [0,0,0],
              [0,1,0]], dtype=np.float32)

dst1 = np.zeros((img.shape[0]-3+1, img.shape[1]-3+1))

for n in range(1, img.shape[1]-1):
    for m in range(1, img.shape[0]-1):
        window = img[m-1:m+2, n-1:n+2]
        gy = np.sum(window * kgy)
        gx = np.sum(window * kgx)

        magnitude = np.sqrt(np.square(gx) + np.square(gy))
        direction = np.arctan(gy/gx)
        if direction < 0:
            dst1[m-1,n-1] = -magnitude
        else:
            dst1[m-1,n-1] = magnitude

#########################################################################
#######
#                              END OF YOUR CODE
#
#########################################################################
#######

# You can achieve a similar (NOT identical) output with the following
code line.
K = np.array([[0, 1,0],
              [1,-4,1],
              [0, 1,0]], dtype=np.float32)
dst2 = cv2.filter2D(img,-1,K)


plt.figure(figsize=(20,10))
plt.subplot(131),plt.imshow(img, 'gray')
plt.title('Original'),plt.xticks([]),plt.yticks([])
plt.subplot(132),plt.imshow(np.abs(dst1), 'gray')
plt.title('Output 1'),plt.xticks([]),plt.yticks([])
plt.subplot(133),plt.imshow(np.abs(dst2), 'gray')
```

```
plt.title('Output 2'),plt.xticks([]),plt.yticks([])
plt.show()
```

```
C:\Users\Admin\AppData\Local\Temp\ipykernel_14824\1704938800.py:23:
RuntimeWarning: invalid value encountered in float_scalars
  direction = np.arctan(gy/gx)
C:\Users\Admin\AppData\Local\Temp\ipykernel_14824\1704938800.py:23:
RuntimeWarning: divide by zero encountered in float_scalars
  direction = np.arctan(gy/gx)
```



Histogram

- It is a graphical representation of the intensity distribution of an image.
- It quantifies the number of pixels for each intensity value considered.

Histogram equilization

- Equalization implies mapping one distribution (the given histogram) to another distribution (a wider and more uniform distribution of intensity values) so the intensity values are spreaded over the whole range.
- To accomplish the equalization effect, the remapping should be the cumulative distribution function (cdf) (more details, refer to Learning OpenCV). For the histogram H(i), its cumulative distribution $H'(i)$ is:

$$H'(i) = \sum_{0 \le j < i} H(j)$$

- To use this as a remapping function, we have to normalize $H'(i)$ such that the maximum value is 255 (or the maximum value for the intensity of the image). From the example above, the cumulative function is:

cumulative distribution function

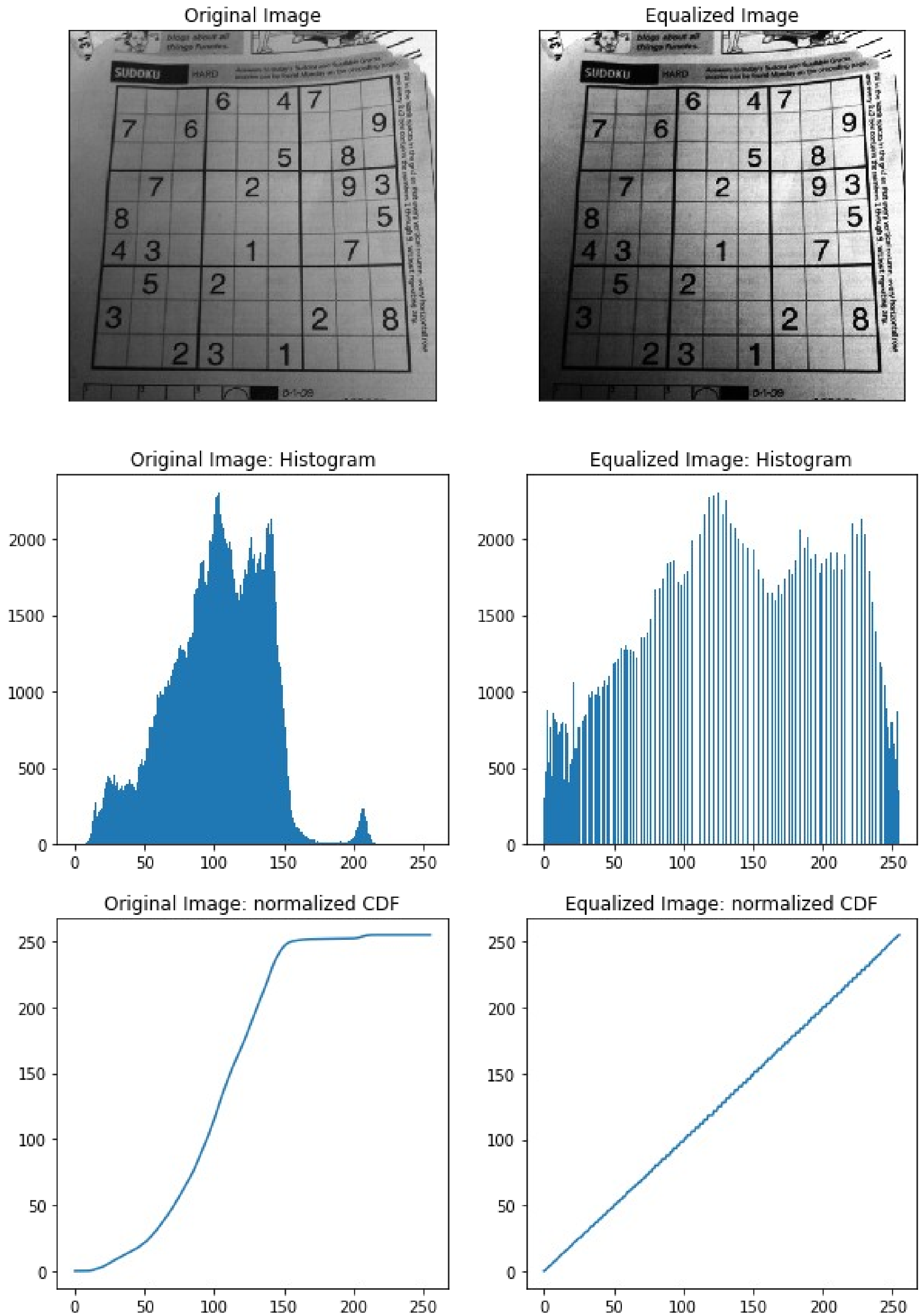
- Finally, we use a simple remapping procedure to obtain the intensity values of the equalized image:

$$equalized(x,y) = H'(src(x,y))$$

Histogram Equalization

```
img = cv2.imread('imgs/sudoku-original.jpg',0)
W,H = img.shape
```

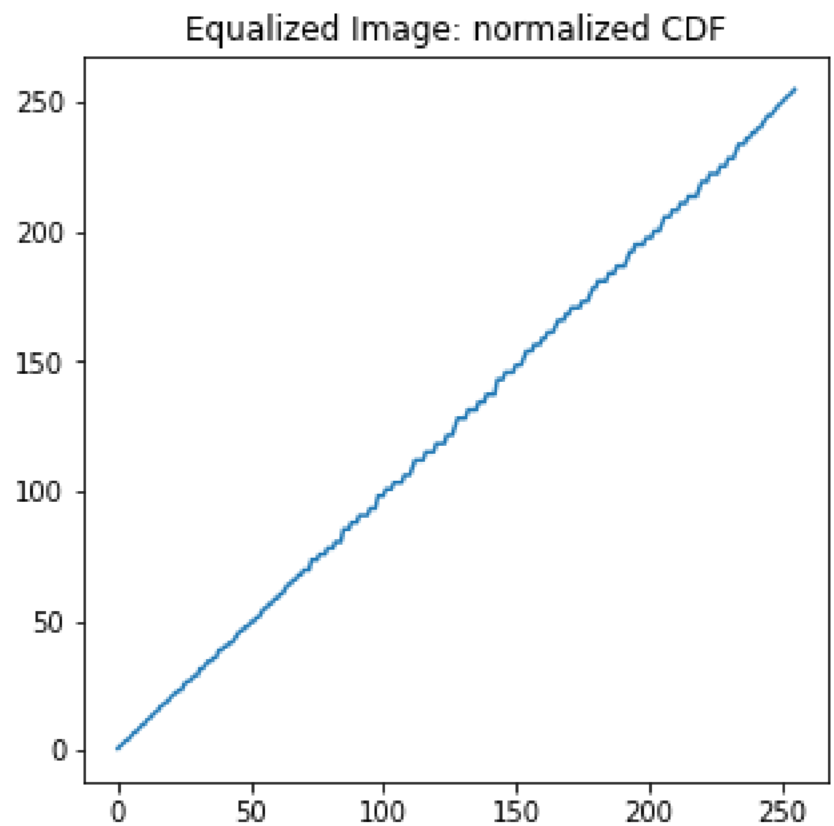
```
img_eq = cv2.equalizeHist(img)

hist = np.histogram(img, bins=256, range=(0.0, 255.0))
hist_eq = np.histogram(img_eq, bins=256, range=(0.0, 255.0))

plt.figure(figsize=(10,15))
plt.subplot(321),plt.imshow(img, cmap='gray'),plt.title('Original
Image'),plt.xticks([]),plt.yticks([])
plt.subplot(322),plt.imshow(img_eq, cmap='gray'),plt.title('Equalized
Image'),plt.xticks([]),plt.yticks([])
plt.subplot(323),plt.hist(img.ravel(), bins=256, range=(0.0,
255.0)),plt.title('Original Image: Histogram')
plt.subplot(324),plt.hist(img_eq.ravel(), bins=256, range=(0.0,
255.0)),plt.title('Equalized Image: Histogram')
plt.subplot(325),plt.plot(range(0,256),np.cumsum(hist[0])*255/(W*H)),p
lt.title('Original Image: normalized CDF')
plt.subplot(326),plt.plot(range(0,256),np.cumsum(hist_eq[0])*255/(W*H)
),plt.title('Equalized Image: normalized CDF')
plt.show()
```



QUIZ: Is histogram equalization reversible?

Your answer: Histogram equalization is not reversible

```
def myEqualizeHist(img):
    """
    A implementation of a histogram equalization for image of `uint8`
data type.
    """
    out = img

######################################################################
#######
    # TODO: Implement the histogram equalization function.
#

######################################################################
#######
    # get image histogram
    image_histogram, bins = np.histogram(img.flatten(), 256,
density=True)
    cdf = image_histogram.cumsum()
    cdf = 255 * cdf / cdf[-1] # normalize

    # linear interpolation of cdf to find new pixel values
    image_equalized = np.interp(img.flatten(), bins[:-1], cdf)

    out = image_equalized.reshape(img.shape)


######################################################################
#######
    #                             END OF YOUR CODE
#

######################################################################
#######
    return out

# Verify the correctness of your implementation by plotting the
# normalized CDF of equalized image
img = cv2.imread('imgs/sudoku-original.jpg',0)
W,H = img.shape
img_myeq = myEqualizeHist(img)

# Your implementation may NOT need to return an image that is
# exactly the same as the one OpenCV build-in function does.
# However, the normalized CDF should make sense.
hist_myeq = np.histogram(img_myeq, bins=256, range=(0.0, 255.0))
plt.figure(figsize=(5,5))
plt.plot(range(0,256),np.cumsum(hist_myeq[0])*255/(W*H))
plt.title('Equalized Image: normalized CDF')
plt.show()
```



Threshold

Simple Threshold

If pixel value is greater than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black). The function used is cv2.threshold.

```
# Get list of available flags for thresholding styles
flags = [i for i in dir(cv2) if i.startswith('THRESH_')]
print(flags)
```

```
['THRESH_BINARY', 'THRESH_BINARY_INV', 'THRESH_MASK', 'THRESH_OTSU',
'THRESH_TOZERO', 'THRESH_TOZERO_INV', 'THRESH_TRIANGLE',
'THRESH_TRUNC']
```

Adaptive Method

It decides how thresholding value is calculated. The function used is cv2.adaptiveThreshold.

- cv2.ADAPTIVE_THRESH_MEAN_C : threshold value is the mean of neighbourhood area.

- cv2.ADAPTIVE_THRESH_GAUSSIAN_C : threshold value is the weighted sum of neighbourhood values where weights are a gaussian window.

```
img = cv2.imread('imgs/sudoku-original.jpg',0)
img = cv2.medianBlur(img,5)
img_mean = np.mean(img)

C = 2
ret,th1 = cv2.threshold(img,img_mean,255,cv2.THRESH_BINARY)
th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C,\
            cv2.THRESH_BINARY,11,C)

th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\
            cv2.THRESH_BINARY,11,C)

######################################################################
#######
# TODO:                                                              #
# Trying several value of constant C and observing how the output
#
# thresholded images change.
#
######################################################################
#######
def test(val):
    C = val
    ret,th1 = cv2.threshold(img,img_mean,255,cv2.THRESH_BINARY)
    th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C,\
                cv2.THRESH_BINARY,11,C)

    th3 =
cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\
                cv2.THRESH_BINARY,11,C)
    return ret, th1, th2, th3

ret, th1, th2, th3 = test(20)

######################################################################
#######
#                              END OF YOUR CODE
#
######################################################################
#######

titles = ['Original Image', 'Global Thresholding (v =
{:.2f})'.format(img_mean),
        'Adaptive Mean Thresholding', 'Adaptive Gaussian
Thresholding']
images = [img, th1, th2, th3]
```

```
fig = plt.figure(figsize=(10, 10))
for i in range(4):
    plt.subplot(2,2,i+1)
    plt.imshow(images[i],'gray')
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```