# Computer Vision
# Degree in Information Technology
# 2º Semester 2021/2022

## Worksheet 3

**Goals:**

- Image processing – arithmetic and logic operations
- Histograms

# Exercises

# Part I – Arithmetic operations

1- Write and run the following code in a python file:

```
import numpy as np
import cv2

# Load the image and show it
image = cv2.imread("DCT_img.jpg")
cv2.imshow("image",image)
cv2.waitKey(0)

# OpenCV arithmetic operations
print("max of 255: {}".format(cv2.add(np.uint8([200]), np.uint8([100]))))
print("min of 0: {}".format(cv2.subtract(np.uint8([50]),
np.uint8([100]))))

# NumPy arithmetic operations
print("wrap around: {}".format(np.uint8([200]) + np.uint8([100])))
print("wrap around: {}".format(np.uint8([50]) - np.uint8([100])))
```

a) Let us examine the code:
b) Import your packages
c) Images are NumPy arrays, stored as unsigned 8 bit integers. What does this mean? It means that the values of our pixels will be in the range [0, 255]. When using functions like cv2.add and cv2.subtract, values will be clipped to this range, even if the added or subtracted values fall outside the range of [0, 255].
d) If you use NumPy arithmetic operations on these arrays, the values will be modulus (wrap around) instead of being clipped to the [0, 255] arrange.
    a. Once a value of 255 is reached, NumPy wraps around to zero, and then starts counting again, until 100 steps have been reached.
    b. Once 0 is reached, the modulus operations wraps around and starts counting backwards from 255.

2- Write and run the following code in a python file:

```
import numpy as np
import cv2

# Load the image and show it
image = cv2.imread("DCT_img.jpg")
cv2.imshow("image",image)
cv2.waitKey(0)

M = np.ones(image.shape, dtype = "uint8") * 100
added = cv2.add(image, M)
cv2.imshow("add",added)
cv2.waitKey(0)

M = np.ones(image.shape, dtype = "uint8") * 50
subtracted = cv2.subtract(image, M)
cv2.imshow("Subtracted", subtracted)
cv2.waitKey(0)
```

a) Let us examine the code:
b) We defined a NumPy array of ones, with the same size as our image. We use 8-bit unsigned integers as our data type. To fill our matrix with values of 100's rather than 1's, we simply multiplied our matrix of 1's by 100.
c) We used the cv2.add function to add our matrix of 100's to the original image. The values were clipped or wrapped?
d) Adding: Used to blend the pixel contents from two images; the resulted image is brighter than the original.
e) We defined another NumPy array of ones.
f) We used the cv2.subtract function to subtract 50 from each pixel intensity of the image.
g) Subtraction: Used to detect differences between two images, decrease its overall brightness, or obtain its negative.

# Part II– Logic operations

1- Write and run the following code in a python file

```
import numpy as np
import cv2

# draw a rectangle
rectangle = np.zeros((300, 300), dtype = "uint8")
cv2.rectangle(rectangle, (25, 25), (275, 275), 255, -1)
cv2.imshow("Rectangle", rectangle)

# draw a circle
circle = np.zeros((300, 300), dtype = "uint8")
cv2.circle(circle, (150, 150), 150, 255, -1)
cv2.imshow("Circle", circle)

bitwiseAnd = cv2.bitwise_and(rectangle, circle)
cv2.imshow("AND", bitwiseAnd)
cv2.waitKey(0)

bitwiseOr = cv2.bitwise_or(rectangle, circle)
cv2.imshow("OR", bitwiseOr)
cv2.waitKey(0)

bitwiseXor = cv2.bitwise_xor(rectangle, circle)
cv2.imshow("XOR", bitwiseXor)
cv2.waitKey(0)

bitwiseNot = cv2.bitwise_not(circle)
cv2.imshow("NOT", bitwiseNot)
cv2.waitKey(0)
```

a) Let us examine the code:
b) Import the packages that you will need.
c) We initialize our rectangle image creating a 300X300 NumPy array and then draw a 250X250 white rectangle at the centre of the image.
d) We initialize our circle image creating a 300X300 NumPy array and then draw a white circle with radius of 150 pixels at the centre of the image.
e) AND: A bitwise AND is true if and only if both pixels are greater than zero.
f) OR: A bitwise OR is true if either of the two pixels are greater than zero.
g) XOR: A bitwise XOR is true if and only if either of the two pixels are greater than zero, but not both.
h) NOT: A bitwise NOT inverts the “on” and “off” pixels in an image.

# Part III– Masking

1- Write and run the following code in a python file

```
image = cv2.imread("DCT_img.jpg ")
cv2.imshow("Original", image)

# Construct a mask with a 150x150 square at the centre of it
mask = np.zeros(image.shape[:2], dtype = "uint8")
(cX, cY) = (270, 110)
cv2.rectangle(mask, (cX - 75, cY - 75), (cX + 75 , cY + 75), 255,-1)
cv2.imshow("Mask", mask)

# Apply the mask to the image
masked = cv2.bitwise_and(image, image, mask = mask)
cv2.imshow("Mask Applied to Image", masked)
cv2.waitKey(0)
```

a) Let us examine the code:
b) Import the packages, load the image.
c) Construct a NumPy array, filled with zeros, with the same width and height of the image.
d) Apply the mask using the cv2.bitwise_and function. The AND function will be True for all pixels in the image. Look to the keyword mask. By supplying a mask, the cv2.bitwise_and function only examines pixels that are "on" in the mask. In this case, only pixels that are part of the white rectangle.

2- Write and run the following code in a python file:

a) We can split the channels using cv2.split function.
b) We can merge the channels back together again using the cv2.merge function.
c) An alternative method is to construct a NumPy array of zeros, with the same width and height as the original image, and then to construct the Red channel representation of the image, make a call to cv2.merge, specifying zeros array for the Green and Blue channels. Take similar approaches to the other channels.

```
import cv2import
import numpy as np

image = cv2.imread("DCT_img.jpg ")
cv2.imshow("Original", image)

(B, G, R) = cv2.split(image)
cv2.imshow("Red", R)
cv2.imshow("Green", G)
cv2.imshow("Blue", B)
cv2.waitKey(0)

merged = cv2.merge([B, G, R])
cv2.imshow("Merged", merged)
cv2.waitKey(0)

cv2.destroyAllWindows()
zeros = np.zeros(image.shape[:2], dtype = "uint8")
cv2.imshow("Red", cv2.merge([zeros, zeros, R]))
cv2.imshow("Green", cv2.merge([zeros, G, zeros]))
cv2.imshow("Blue", cv2.merge([B, zeros, zeros]))
cv2.waitKey(0)
```

3- This experiment will serve as curiosity. Write and run the following code:

```
import cv2
image = cv2.imread("DCT_img.jpg ")
cv2.imshow("Original", image)

# RGB color space to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Gray", gray)

# RGB color space to HSV
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
cv2.imshow("HSV", hsv)

#RGB color space to lab
lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
cv2.imshow("L*a*b*", lab)
cv2.waitKey(0)
```

# Part IV – Histograms

1- Write and run the following code in a python file:

```
from matplotlib import pyplot as plt
import cv2
image = cv2.imread("DCT_img.jpg")

#Convert the image from the RGB colorspace to grayscale.
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
plt.imshow(gray, cmap='gray')
plt.title('Gray')
plt.show()

# Construct a grayscale histogram
hist = cv2.calcHist([gray], [0], None, [256], [0, 256])
plt.figure()
plt.title("Grayscale Histogram")
plt.xlabel("Bins")
plt.ylabel("# of Pixels")
plt.plot(hist)
plt.xlim([0, 256])
plt.show()
```

a) Let us examine the code:
b) First you imported the packages that you need.
c) Then you convert the image from the RGB Colo space to grayscale.
d) Function cv2.calcHist computes the actual histogram:

`cv2.calcHist(images,channels,mask,histSize,ranges)`

- images: This is the image that we want to compute a histogram for.
- channels: This is a list of indexes, where we specify the index of the channel we want to compute a histogram for. To compute a histogram of a grayscale image, the list would be [0]. To compute a histogram for all three red, green, and blue channels, the channels list would be [0,1,2].
- mask: If a mask is provided, a histogram will be computed for masked pixels only.
- If we do not have a mask or do not want to apply one, we can just provide a value of None.
- histSize: This is the number of bins we want to use when computing a histogram. Again, this is a list, one for each channel we are computing a histogram for.
- ranges: Here we specify the range of possible pixel values. Normally, this is [0, 256] for each channel.
- **Interpreting the plot:**
    - the bins (0-255) are plotted on the x-axis.
    - the y-axis counts the number of pixels in each bin.

2- Write and run the following code in a python file:

```
from matplotlib import pyplot as plt
import cv2
img = cv2.imread('ponte.jpg')
cv2.imshow("Colour Image", img)
#Split chanels
canais = cv2.split(img)
cores = ("b", "g", "r")
plt.figure()
plt.title("'Colour Histogram")
plt.xlabel("Intensity")
plt.ylabel("Pixels")
#loop for each chanel
for (canal, cor) in zip(canais, cores):
    hist = cv2.calcHist([canal], [0], None, [256], [0, 256])
    plt.plot(hist, cor)
    plt.xlim([0, 256])
plt.show()
```

It is possible to represent the histogram of the three channels at the same time using `zip` function.

3- Write and run the following code in a python file

```
from matplotlib import pyplot as plt
import cv2
import numpy as np
image = cv2.imread("dark.jpg")
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
eq = cv2.equalizeHist(gray)
cv2.imshow('Histogram Equalization (Before and After)', np.hstack([gray,
eq]))
cv2.waitKey(0)
```

**<u>Ideas to the Practical work:</u>**

1- Define a function to plot a histogram for a colour image. It should have three input parameters: the image, the title, the mask. Give the default None value for the mask.

2- Create a python script to load an image which you like. Create a white circle as a mask. Apply the mask with an AND bitwise function to the image.

**3-** A company wants to add its logo to all product images. Using the techniques of image processing (logic, arithmetic, and masking), define a function which add logos to images.


4- Pick an image an generate the corresponding histogram. Do a descriptive analysis about the quality. Then, use a dark image as the following example, originate the histogram, and perform the equalization. Report the results and present the main conclusions.