# *Exercise 2*

## *Introduction to fundamentals of CV in Python*

*IAML 2020*

### *Purpose*

These exercises aim to introduce you to Python, Numpy, and OpenCV. You will make Python scripts to convert images in different color spaces, to change the image properties using point processing, to calculate and plot histrograms from grayscale and multi-channel images, and to select objects using binary masking. The exercises are rather "hand-held" but at the end of this document is an extra exercise. You are NOT expected to do this. However, if you want a challenge of a real world inspired image search problem and want less hand-holding, then please GO FOR IT. It is highly rewarding and fun.

### *Notes*

- In the exercises based on script (i.e. `.py`) files, you are expected to implement code at certain points in the file. Although it is typically clear from context which piece of code you have to change, we have included specifier comments of the type `# <Exercise x.x (n)>`, where `x.x` is the exercise number and `(n)` is the letter associated with the specific task.
- Exercises marked by $^{extra}$ should only be attempted after completing the other exercises. The mark is added to indicate that the exercise is not essential but still important.

**Exercise 2.1**

## *Introduction to Python and Numpy*

The first exercise aims to introduce you to a bit more Python syntax as well as the Numpy library. This exercise is implemented as a Jupyter notebook to provide a more interactive experience. We will generally be using notebooks for exercises that rely heavily on explanations and experimentation.

Notebooks are great for personal experimentation and notes as well.

The instructions for starting Jupyter Lab are:

1. Open a terminal (or Anaconda Prompt on windows) and activate the *iaml* environment using the command `conda activate iaml`.
2. Navigate to the folder containing the exercise material using `cd <folder path>`.
3. Start Jupyter Lab by running the command `jupyter lab`. This should automatically open Jupyter in a browser window.

Then simply open the file `practical_introduction.ipynb` and follow the instructions there.

**Exercise 2.2**

## *Color spaces*

In this exercise you will use `cv2.cvtColor()` [1] to convert images between color spaces. `cv2.cvtColor()` has two mandatory arguments, namely: `src`, the input image; and `code`, the color space conversion code. `cv2.cvtColor()` returns the converted image. You can get more information about a method or function by typing the `help` command the Python Interpreter. For example: `help(cv2.cvtColor)`.

[1] Click here to read the documentation about cv2.cvtColor() in OpenCV.

1. Open `color_spaces.py` from the supporting material and read through the code. Right now, the script only imports Numpy and OpenCV and defines a `filename` variable.
2. Load the input image given by `filename` using `cv2.imread()`.
3. **Conversion between color spaces:** Notice, OpenCV loads color images in BGR format (as opposed to RGB). Covert the input image to both RGB and HSV color spaces (e.g. yielding two different output images) by using the following values for `code`.
   (a) `cv2.COLOR_BGR2RGB` to convert the input image to RGB color space.
   (b) `cv2.COLOR_BGR2HSV` to convert the input image to the HSV color space
   (c) Display the images using the OpenCV function `cv2.imshow()`. The function takes a window name and an image as parameters. This makes it possible to create one window for each image by giving them different names.[2]
4. **Splitting the channels:** This exercise is about splitting an image into individual color channels (e.g. `channel_r`, `channel_g` and `channel_b` for the RGB image) using both :
   1. Python Slicing (Numpy) (e.g. `image_rgb[:, :, 0]`) to get all columns and all rows from red channel). Specifically, t
   2. `cv2.split()` [3]. This method has one argument (i.e. `image`) and it returns each color channel into a tuple of single-channel images.

   Split the RGB image into the R,G, and B components and display them in separate figures/windows as grayscale images.
5. **Change the channel representation:** In the following you will investigate how each component contributes to the color image (e.g. by changing the channels). The following instructions show how to change the green channel representation from a RGB image.
   (a) Use `np.zeros()` [4] to create an array of the same width and height as the channel images. Use `np.zeros(<shape>, dtype=np.uint8)` for creating an array with unsigned 8-bit integer elements.

[2] Reusing the same name simply updates the image in the existing window. This is very useful for videos and image manipulation applications.

[3] Press here to get the help for the OpenCV function `cv2.split()` e.g. that divides a multi-channel array into a tuple of single-channel arrays.

[4] A Numpy function to create an array with all elements equal to zero.

**Hint:** *Remember from Exercise 2.1 that* `array.shape` *can be used to get the shape of an array and* `l[:2]` *returns the first two elements of a list.*

(b) Create one BGR image for each channel with the channel data in the correct position using `cv2.merge()` [5] (e.g. fill out the channels *red* and *blue* with zeros and merge them with the *green* single-channel: `cv2.merge([single, channel_g, single])`).

(c) Use `cv2.imshow()` to show the generated images for single channels. The expected output is shown in Figure 1.

[5] An OpenCV function to create one multi-channel array out of several single-channel ones.


Figure 1: Example of RGB representation.

6. .$^{(extra)}$ Make the same figures as in the previous question but by using assignment
e.g. `A[:,:,0]=SomeGrayImage`).

**Exercise 2.3**

*Point-processing*

This exercise introduces you to point-processing in OpenCV as well as a possible skeleton for simple GUI applications using OpenCVs API. You will implement point-processing methods for brightness, contrast, and image negation. Use the file `point_processing.py` available in the supporting material to develop the exercise.

1. Open the file `point_processing.py` and run the program. Notice the image window with trackbars. We will add functionality to these next.

2. **GUI:** The trackbars are created in the `setup_trackbars` method. Each trackbar is created using `cv2.createTrackBar()` [6] which automatically adds it to the OpenCV window with the specified name. You will find this function useful later when creating tests for your own image applications.

   For now, we only concentrate on the last argument given to the trackbar function. The last argument is a reference to a function which is called whenever the trackbar is updated, i.e. when it is dragged by a user. Each of the three bars `Contrast`, `Brightness`, and `Negative`, have a method associated with it.

   Taskbars output only integer values between zero and whichever maximum is passed to `cv2.createTrackBar()`'s 4th argument. Your task is to map the trackbars integer outputs to the following ranges:

   - `Contrast:` Between $[0, 2]$
   - `Brightness:` Between $[-128, +128]$
   - `Negative:` 0 OR 255

   You do this by changing each of the three trackbar update methods `change_contrast()`, `change_brightness()` and `change_inverted()` to update the corresponding class fields `self.contrast`, `brightness`, and `self.inverted`.

3. **Contrast:** The contrast describes the level of details we can see in an image. Change the method `apply_contrast()` so that when given an input image $f$ it returns an output image $g$ with different contrast. Use the equation $g(x, y) = a \times f(x, y)$ to change the contrast.

   (a) The contrast equation can produce values larger than 255. You have to replace all values larger than 255 to 255 (white pixel).

   (b) Convert the `g` variable to 8-bit unsigned integers using `np.uint8()`.[7].

4. **Brightness:** The brightness describes the overall intensity levels of an image. Change the method `apply_brightness()` so that

[6] Link to docs.

**Hint:** *Use comparison operators to easily find and replace the values* $> 255$. *See Exercise 2.1.4 under* comparison operators.

[7] List of supported Numpy array types and conversions.

it given an input image $f$ and $b$ returns $g(x,y) = f(x,y) + b$. Like for the previous question for contrast, make sure $g$ is in the range [0-255] and has type `uint8`.

5. **Negative:** The negative of a grayscale level from an individual pixel. Develop a function called `apply_inverted()` that calculates the inverted image. Use the equation $g(x,y) = 255 - f(x,y)$. Again, make sure that `g` is of type `uint8`.

6. **Point Processing:** Implement the function `point_processing()`. It should apply each of the point-processing operations to the input and return the result.

7.

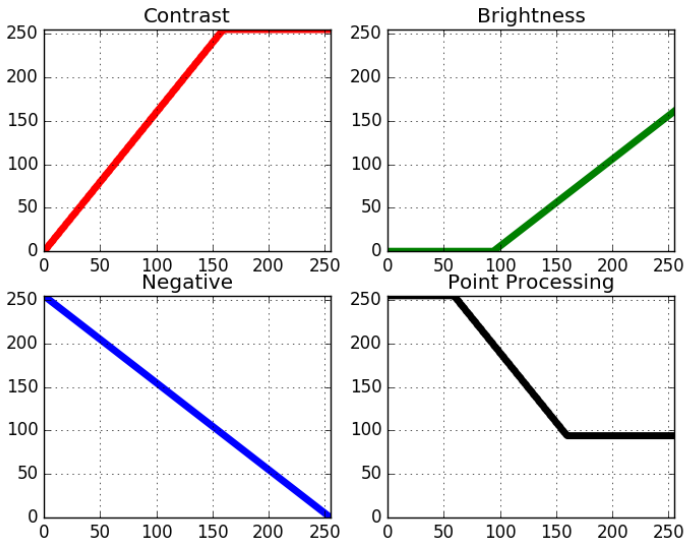



Figure 2: This Matplotlib window shows the point processing transformations.

**Exercise 2.4**

## *Using colors for Detection*

In this exercise, you will detect pixels belonging to objects based on colors in both images and videos. Figure 3 show an example of detecting a *blue notebook*. Use the file `detector.py` available in the supporting material to make the exercise.

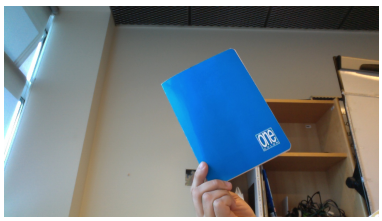


Figure 3: Object to be detected in the current frame.

(a) **Convert to RGB and HSV:** You will experiment with image processing using both color spaces. Each colorspace has its pros and cons. Use the function `cv2.cvtColor()` to convert the input image in both HSV and RGB color spaces.

(b) **Create the Mask:** This object detection algorithm is based on binary mask concept. The ideas is to create a binary mask to select the regions of interest in the input image and to update this mask for each new frame. Instead of creating a white rectangle or a white circle, your implementation needs to identify the pixels where the object is located in the current frame.

Use the function `cv2.inRange()` [8] to create a binary mask between a range of color (i.e. using the variables `lower` and `upper`). Use the current HSV converted image as input and then you have to specify the `lower` color, and the `upper` color. Check in this link (https://goo.gl/xKFxFi) how to find a good HSV values to track.

[8] An OpenCV function to check if array elements lie between the elements of two other arrays.

(c) **Apply the Mask:** Use the function `cv2.bitwise_and()` to select the pixels of interest in the image using the binary mask.

(d) **Detect Additional Objects:**$^{extra}$ Change the `lower` color and the `upper` color to detect *Red* objects, *Green* objects, your *clothes*, your *skin*, among others.

(e) **Don't use OpenCV for masking**$^{extra}$ Try to perform the functionality of `cv2.inRange()` and `cv2.bitwise_and()` using just Numpy.

**Hint:** *All necessary syntax and methods are mentioned in Exercise 1.*

Figure 4 shows the color detection results in a single image. The video result of the using Python and OpenCV on https://youtu.be/kipIUbtM5mo.

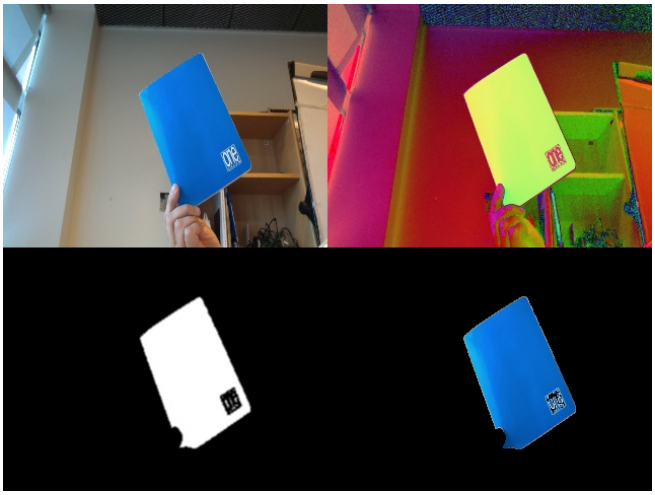


Figure 4: Steps of color-based object detection: (top left) the original input image, (top right) the HSV image, (bottom left) the mask created from the range of colors, (bottom right) and the final detection result.

**Exercise 2.5**

## *Histograms*

In this exercise, you will calculate the histograms of a grayscale image, a shuffled image and multi-channel images. You will use the file `histogram.py` from the exercise material as a starting point.

Inspect the functions `show_image()` and `show_histogram` as well as the two variables `fig_imgs` and `fig_hist`. The functions help simplify plotting multiple images and histograms. You will use these to visualise the results for each part of this exercise.

(a) **Grayscale Histogram:** Use the function `cv2.calcHist()` [9] to calculate the histogram of the grayscale image (contained in the variable `grayscale`). `cv2.calcHist()` has the following parameters:

- `images`: the source of arrays or images. Use a list of images to get a n-dimensional histogram of each channel e.g. `[image[0], image[1]]`. For gray scale images use. `[grayscale]`.
- `channels`: a list of the channels used to compute the histogram. For a single-channel histogram use `[0]`, and for multiple-channel histogram use `[0, 1]`;
- `mask`: an optional mask. A mask is used to select a region of interest in the processed image. For this exercise, use `None`;
- `histSize`: an array of histogram sizes in each dimension. As you will work with grayscale images (or each individual color channels), use the value 256 to calculate a histogram between 0 and 255. For a single histogram use the code `[256]`, and for $n$-dimension histogram use `[256, 256]`; and
- `ranges`: an array of the histogram bin boundaries in each dimension to specify the range of values. For a single histogram use the code `[0, 255]`, and for $n$-dimension histogram use `[0, 255, 0, 255, 0, 255]`.

[9] Link to in-depth guide.

Visualise the grayscale image and corresponding histogram using the aforementioned helper functions.

(b) **Shuffled Grayscale Histogram:** Use the function `numpy.random.shuffle()` [10] to shuffle the elements of rows or columns of an input image. Calculate the histogram of the shuffled image and visualise both.

[10] Link to docs.

(c) Make a function `calculate_histogram_distance(hist1,hist2)` which returns the squared Euclidian distance between the two histograms, i.e.

$$\|H^a - H^b\|^2 = \sum_{i=1}^{N} (H_i^b - H_i^b)^2, \tag{1}$$

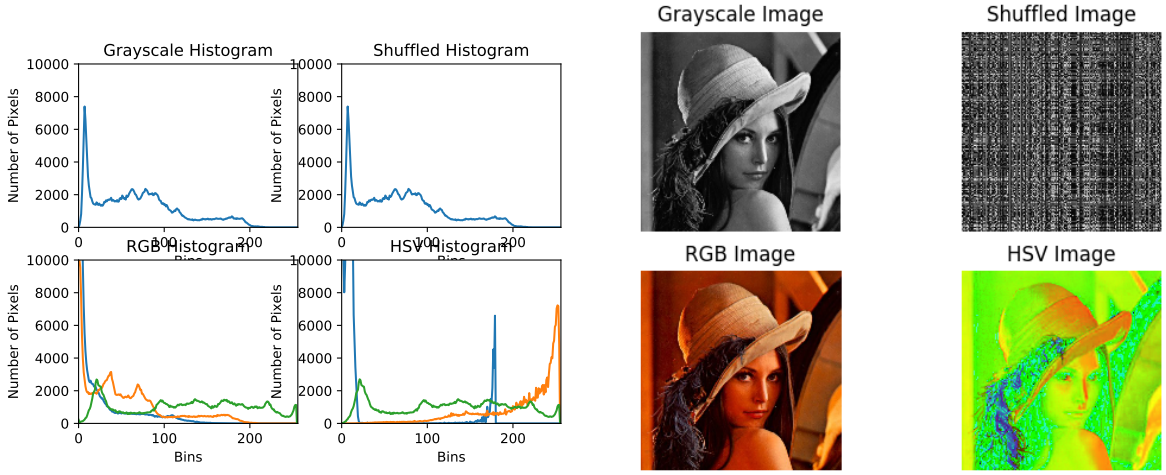
where $H_i$ is the $i$'th bin of histogram $H$.

(d) Calculate the distance between the histograms of the original image and the shuffled image and print the result. Is this what you expected? Can you explain why this is the case?

(e) **RGB Histogram:** Use the input image to calculate the histogram from each individual image channel. Vectorize the histograms of each RGB channel into one Numpy array ($256 \times 3$ Numpy array). Visualise the results. How is the histogram different from the grayscale histogram? Why do you think this is?

(f) **HSV Histogram:** Convert the input image (or RGB image) in a HSV image. Calculate the histogram from each individual channel and vectorize them into one Numpy array ($256 \times 3$). Visualise the results. Can you explain what causes the difference between the HSV and RGB histograms?

The end result should be similar to the following image:



**Exercise 2.6**

*Masking*$^{extra}$

In this exercise, you will be experimenting with masks and logical operators on images. You will use the file `masking.py` available in the supporting material to develop your Python code.

(a) **Create the Mask:** Use the function `numpy.zeros()` to create a single-channel Numpy array with the same resolution of the input image. *Tips:* Remember to define its type as unsigned integer ( `dtype=np.uint8` ).

(b) **Masking a Rectangle Object:** Use the function `cv2.rectangle()` [11] to draw two rectangles in the mask image. This function has the following parameters:

- `img` : input image (or mask);
- `pt1` : vertex of the rectangle. Set the upper left coordinate in the mask;
- `pt2` : vertex of the rectangle. Set the bottom right coordinate in the mask;
- `color` : rectangle color or brightness (grayscale image). Set the white color; and
- `thickness` : thickness of lines that make up the rectangle. Set the thickness as $-1$ (or any negative number) to draw a filled white rectangle in the mask.

[11] Link to OpenCV shape guide.

(c) **Masking a Circle Object:** Use the function `cv2.circle()` to draw a circle in the mask. This function has the following parameters:

- `img` : image (or mask);
- `center` : center coordinates of the circle;
- `radius` : radius of the circle;
- `color` : circle color or brightness (grayscale image). Set the white color; and
- `thickness` : thickness of the circle outline, if positive. Negative thickness means that a filled circle is to be drawn. Set the thickness as $-1$.

(d) **Applying the Mask:** Use the function `cv2.bitwise_and()` [12,13] to apply a binary mask to the RGB image, and select only the region of interest on it. This function uses the operator $AND$ to create a third image with the values available in both input images (i.e. parameters `src1` and `src2` ). The function `cv2.bitwise_and()` has a third parameter to select only the pixels that lie within the white region in the mask. Use the following command to apply the mask in the RGB image:

```
result = cv2.bitwise_and(image, image, mask=mask_img)
```

[12] Link to guide with further details

[13] Link to docs.

Figure 5 shows an example of masking the face of two football players and the ball. This image (called `zico.jpg`) is available in the supporting material.

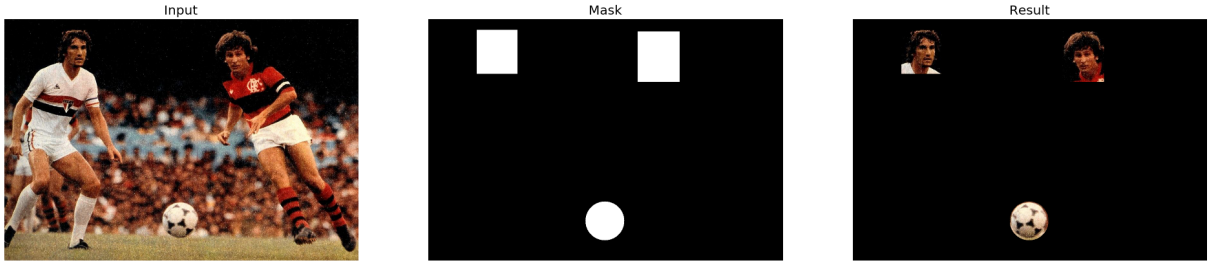



Figure 5: Final result of masking processing.