# Assignment 1: Basic image processing and histograms

Machine perception

2022/2023

Create a folder assignment1 that you will use during this assignment. Unpack the content of the assignment1.zip that you can download from the course webpage to the folder. Save the solutions for the assignments as Python scripts to assignment1 folder. In order to complete the assignment you have to present your solutions to the teaching assistant. Some assignments contain questions that require sketching, writing or manual calculation. Write these answers down and bring them to the assignment defense as well.

The maximum total amount of points for each assignment is 100. **All** the tasks in the assignment are mandatory, except if they are marked with ★ . You can obtain 75 points for solving all the mandatory tasks. The remaining 25 points can be gained by solving optional tasks. There might be more than 25 points for optional assignments. In that case, the maximum of 100 points can **not** be exceeded, even if you solve all of them. The number of points for each optional task is written next to the instructions.

At the defense, the obligatory tasks **must** be implemented correctly and completely. If your implementation is incomplete, you will not be able to successfully defend the assignment.

---

The purpose of this assignment is to familiarize yourself with the working of *NumPy* and *OpenCV* libraries that will be used for the practical part of this course. This assignment will cover reading image data into matrices, manipulating parts of images or channels, describing images with histograms, basic thresholding, and morphological operations.

## Exercise 1: Basic image processing

In the archive assignment1.zip you will find the utility functions that you can use for reading and displaying images in file UZ_utils.py.

(a) Read the image from the file umbrellas.jpg and display it using the following snippet:

```python
from UZ_utils import *
I = imread('image.jpg')
imshow(I)
```

This snippet uses our imread function that reads the image, converts it to floating points and normalizes its values to the interval [0, 1]. If you require images of integer type, you can also read them using `cv2.imread()` or `plt.imread()`. You can import libraries *NumPy*, *OpenCV* and *Pyplot* like in the following snippet:

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

The image is now represented as a *NumPy* array. You can check its size by using the following command `height, width, channels = I.shape`. Just as important is the type of the matrix (in *NumPy* accessed with `I.dtype`). Images are usually loaded as a matrix of type `np.uint8`, which represents unsigned integers with 8 bits, effectively giving the range of `[0,255]`. In this course, only `np.uint8` and `np.float64` types will be used for representing images. The conversion can be performed as follows: `I_float = I.astype(np.float64)`

**Note**: It is a good practice to always convert the image to float data type and scale it to $[0, 1]$. See the file *UZ_ utils.py* for an example and other utilities.

**Note**: Beware of assignment by reference. The command `I_new=I` for *NumPy* arrays only creates a new reference to the same data and does *not* copy it. You can copy the data using `I_new=np.copy(I)`.

**Note**: The library `pyplot` can be used to display image data (function `plt.imshow(I)`). The function `plt.show()` *must* then be called in order for the window to be displayed.

(b) Convert the loaded image to grayscale.[1] A very simple way of doing this is summing up the color channels and dividing the result by 3, effectively averaging the values. The issue, however, is that the sum easily reaches beyond the `np.uint8` range. We can avoid that by casting the data to a floating point type. You can access a specific image channel using the indexing syntax like `red = I[:,:,0]`.

**Note**: If loading images using `cv2.imread()` in fact returns the channel ordering **BGR** instead of **RGB**. This can be fixed using `I = cv2.cvtColor(I, cv2.COLOR_-BGR2RGB)`

(c) Cut and display a specific part of the loaded image. Extract only one of the channels so you get a grayscale image. You can do this by indexing along the first two axes, for instance: `cutout=I[130:260, 240:450, 1]`. You can display multiple images in a single window using `plt.subplot()`.

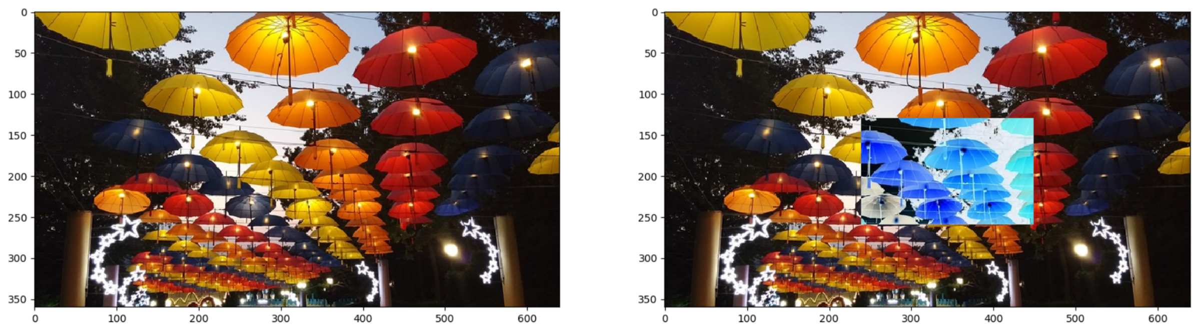
Grayscale images can be displayed using different mappings (on a RGB monitor, every value needs to be mapped to a RGB triplet). *Pyplot* defaults to a color map named *viridis*, but often it is preferable to use a grayscale color map. This can be set with an additional argument to `plt.imshow`, like `plt.imshow(I, cmap='gray')`.

***Question:*** Why would you use different color maps?

(d) You can also replace only a part of the image using indexing. Write a script that inverts a rectangular part of the image. This can be done pixel by pixel in a loop or by using indexing.

***Question:*** How is inverting a grayscale value defined for *uint8*?

[1]When the instructions are given to implement something specifically like here, you are not allowed to use implementations from libraries that do exactly what your task is.



(e) Perform a reduction of grayscale levels in the image. First read the image from `umbrellas.jpg` and convert it to grayscale. You can write your own function for grayscale conversion or use the function in *UZ_utils.py*.

Convert the grayscale image to floating point type. Then, rescale the image values so that the largest possible value is 63. Convert the image back to *uint8* and display both the original and the modified image. Notice that both look the same. *Pyplot* tries to maximize the contrast in displayed images by checking their values and scaling them to cover the entire *uint8* interval. If you want to avoid this, you need to set the maximum expected value when using `plt.imshow()`, like `plt.imshow(I, vmax=255`. Use this to display the resulting image so the change is visible.

## Exercise 2: Thresholding and histograms

Thresholding an image is an operation that produces a binary image (mask) of the same size where the value of pixels is determined by whether the value of the corresponding pixels in the source image is greater or lower than the given threshold.

(a) Create a binary mask from a grayscale image. The binary mask is a matrix the same size as the image which contains `1` where some condition holds and `0` everywhere else. In this case the condition is simply the original image intensity. Use the image `bird.jpg`. Display both the image and the mask.

The binary mask creation can be performed by using the selection syntax of *NumPy* and separately setting all pixels larger and smaller than the threshold to `0` or `1` respectively:

```
threshold = 80

I[I<threshold]=0
I[I>=threshold]=1
```

Alternatively, this can also be done by using `np.where(condition, x, y)`. If the condition holds, the value will be replaced by x, otherwise by y. Write a script that implements both ways. Experiment with different threshold values to obtain a reasonably good mask of the central object in the image.

(b) Setting the threshold manually can be tedious. We will use a representation of the image called a *histogram* to try and set the threshold automatically.

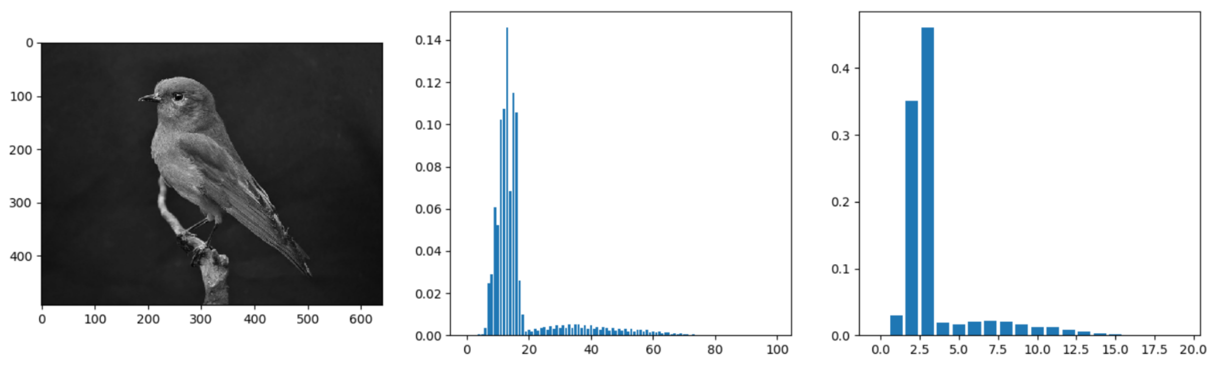
Write a function `myhist` that accepts a grayscale image and the number of bins that will be used in building a histogram. The function should return a 1D array that represents the image histogram (the size should be equal to the number of bins, of course).

The histogram is simply a count of pixels with same (or similar) intensity for all bins. You can assume the values of the image are within the interval `[0,255]`. If you use fewer than `255` bins, intensities will have to be grouped together, e.g. if using 10 bins, all values on the interval `[0,25]` will fall into bin `0`. You can create an empty *NumPy* array with `H = np.zeros(n_bins)`

*Hint:* You may want to use `I.reshape(-1)` to unroll your image into a 1D vector.

***Question:*** The histograms are usually normalized by dividing the result by the sum of all cells. Why is that?

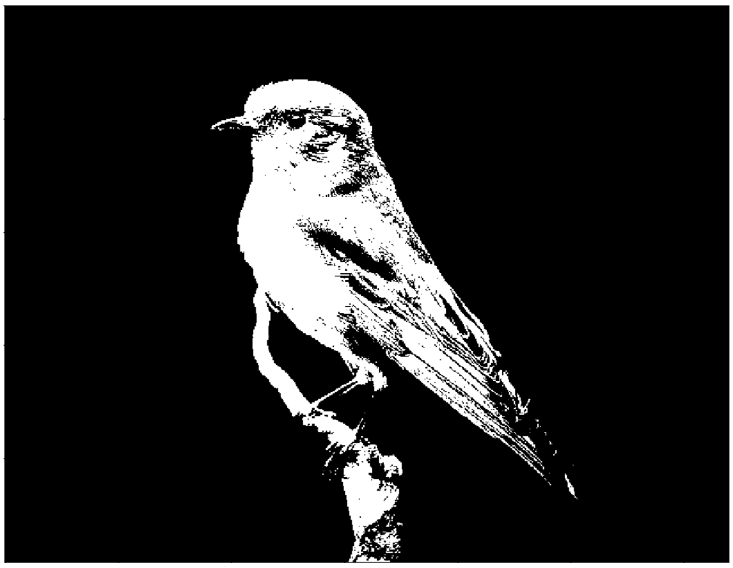
Write a script that calculates and displays histograms for different numbers of bins using `bird.jpg`.



(c) ★ (5 points) Modify your function `myhist` to no longer assume the *uint8* range for values. Instead, it should find the maximum and minimum values in the image and calculate the bin ranges based on these values. Write a script that shows the difference between both versions of the function.

(d) ★ (5 points) Test `myhist` function on images (three or more) of the same scene in different lighting conditions. One way to do this is to capture several images using your web camera and change the lighting of the room. Visualize the histograms for all images for different number of bins and interpret the results.

(e) ★ (15 points) Implement Otsu's method for automatic threshold calculation. It should accept a grayscale image and return the optimal threshold. Using normalized histograms, the probabilities of both classes are easy to calculate. Write a script that shows the algorithm's results on different images.



## Exercise 3: Morphological operations and regions

While thresholding can in some cases give you a good mask of the object, it is still just a global technique that can produce artifacts such as holes in the object or unwanted noise on the background. Such artifacts are best removed before further processing. Morphological operations can be used for removing them.

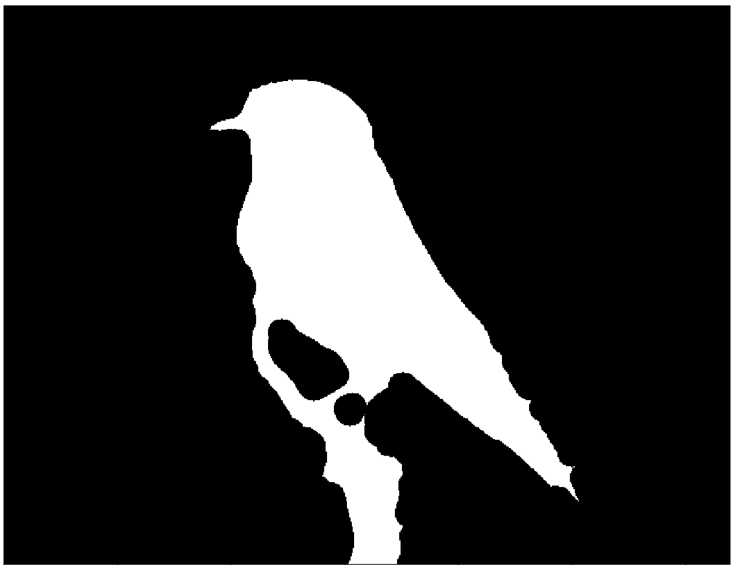
(a) We will perform two basic morphological operations on the image `mask.png`, *erosion* and *dilation*. We will also experiment with combinations of both operations, named *opening* and *closing*.

Use the following snippet and write a script that performs both operations with different sizes of the structuring element. Also combine both operations sequentially and display the results.

```
n = 5
SE = np.ones((n,n), np.uint8) # create a square structuring element
I_eroded = cv2.erode(I, SE)
I_dilated = cv2.dilate(I, SE)
```

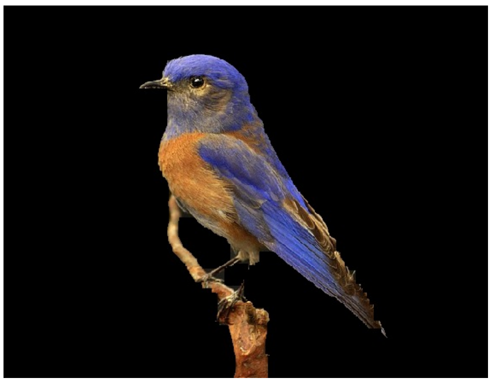
***Question:*** Based on the results, which order of erosion and dilation operations produces *opening* and which *closing*?

(b) Try to clean up the mask of the image `bird.jpg` using morphological operations as shown in the image. Experiment with different sizes of the structuring element. You can also try different shapes, like `cv2.getStructuringElement(cv2.MORPH_-ELLIPSE,(n,n))`.



(c) ★ (5 points) Write a function `immask` that accepts a three channel image and a binary mask and returns an image where pixel values are set to black if the corresponding pixel in the mask is equal to `0`. Otherwise, the pixel value should be equal to the corresponding image pixel. Do not use for loops, as they are slow.

*Hint:* Use `np.expand_dims` function to make the mask and the image to have the same number of axes.



(d) Create a mask from the image in file `eagle.jpg` and visualize the result with `immask` (if available, otherwise simply display the mask). Use Otsu's method if available, else use a manually set threshold.

***Question:*** Why is the background included in the mask and not the object? How would you fix that in general? (just inverting the mask if necessary doesn't count)

(e) Another way to process a mask is to extract connected components. To do this, you can use the function `cv2.connectedComponentsWithStats`[2] that accepts a binary image and returns information about the connected components present in it. Write a script that loads the image `coins.jpg`, calculates a mask and cleans it up using morphological operations. Your goal is to get the coins as precisely as possible. Then, using connected components, remove the coins whose area is larger than `700` pixels from the original image (replace them with white background). Display the results.

[2]Documentation available at https://bit.ly/3uFKYTY

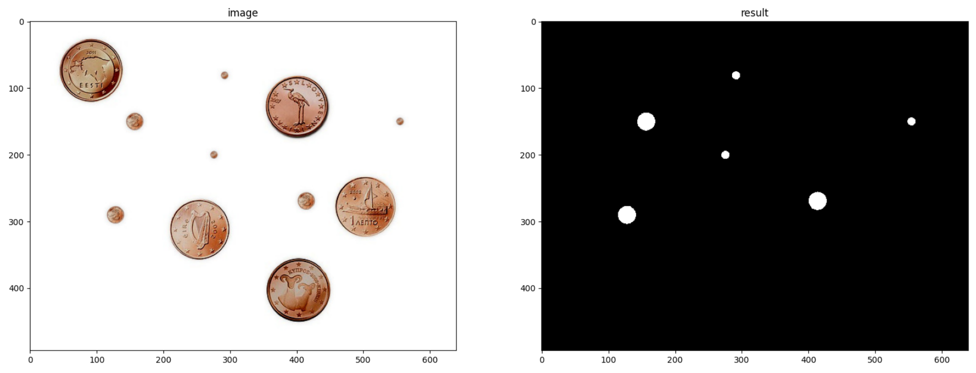


image
result