# Advanced Vision Systems

AGH UST International Course

Tomasz Kryjak, PhD, Mateusz Wąsala, MSc

EMBEDDED VISION SYSTEMS GROUP
VISION SYSTEMS LABORATORY
WEAIIB, AGH UNIVERSITY OF KRAKOW

*Third edition, revised and corrected, Krakow, February 2024*

# Contents

# Laboratory 1

# 1. Vision algorithms in Python 3.X – introduction

The exercise will present/mention basic information related to the use of the Python language to perform image and video processing operations and image analysis.

## 1.1 Programming software

Before starting the exercises, **please create** your own working directory in the place given by the tutor. The name of the directory should be associated with your name – the recommended form is `LastName_FirstName` without no special characters.

The Python programming environment – Visual Studio Code (VSCode) – can be used to complete the exercises. Visual Studio Code is a free, lightweight, intuitive and easy to use code editor. It is a universal software for any programming language. Configuration of the editor for exercises comes down only to choosing the appropriate interpreter, or more precisely the appropriate virtual environment in the Python language.

## 1.2 Python modules used in image processing

Python does not have a native array type that allows to perform elementwise operations betweens two containers (in a convenient and efficient way). For operations on 2D arrays (i.e. also on images) it is common to use the external module NumPy. This is the basis for all further mentioned modules supporting image processing and display. There are at least 3 main packages supporting image processing:

- a pair of modules: the *ndimage* module from the *SciPy* library – contains functions for processing images (including multidimensional images), and the *pyplot* module from the *Matplotlib* library – for displaying images and plots,
- *PILLOW* module (part of the non-expanded PIL module)
- *cv2* module is an frontend to the popular *OpenCV* library.

In our course we will rely on *OpenCV*, although *Matplotlib* and occasionally *ndimage* will also be used – mainly in situations where the functions in *OpenCV* do not have adequate functionality or are less convenient to use (e.g. displaying images).

## 1.3 Input/output operations

### 1.3.1 Reading, displaying and saving an image using OpenCV

**Exercise 1.1** Perform a task in which you practice handling files using OpenCV.

1. From the course page, download the image *mandril.jpg* and place it in your own working directory.
2. Run the program – *spyder*, *pycharm*, *vscode* – from the console or using the icon. Create a new file and save it in your own working directory.
3. In the file, load the module `cv2` (`import cv2`). Test loading and displaying images using this library – use the function `imread` for loading, example usage: `I = cv2.imread('mandril.jpg')`
4. Displaying a picture requires at least two functions – `imshow()` creates a window and starts the display, and `waitKey()` shows the picture for a given period of time (argument 0 means wait for a key to be pressed). When the program is finished, the window is automatically closed, but on condition that it is terminated by pressing any key. Closing the window via the button on the window bar will loop the program and you will have to abort it:
   - VSCode – stop the program in the terminal by using the keyboard shortcut 'CTRL+Z' or closing the terminal.

   R For especially high-resolution images, it is useful to use the function `namedWindow()` with the same name as in the function *imshow()*. We declare it before the *imshow()* function.

5. An unnecessary window can be closed using `destroyWindow` with the appropriate parameter, or all open windows can be closed using `destroyAllWindows`.

```
I = cv2.imread('mandril.jpg')
cv2.imshow("Mandril",I)      # display
cv2.waitKey(0)                # wait for key
cv2.destroyAllWindows()       # close all windows
```

   R The window does not necessarily need to be displayed on top. It is good practice to use the *cv2.destroyAllWindows()* function.

6. Saving to a file is performed by the function *imwrite* (please note the format change from *jpg* to *png*):

```
cv2.imwrite("m.png",I) # zapis obrazu do pliku
```

7. The image displayed is in colour, which means that it consists of 3 channels. In other words, it is represented as a 3D array. There is often a need to view the value of this array. This can be done in one-dimensional form, but is better done with a two-dimensional array.
   - VSCode – it is necessary to run the program in *Debug* mode and set the breakpoint. Under *Run and Debug* and in the *Variables* section you will find the variable *I*. To view the values of the *I* matrix, select *View Value in Data Viewer* under the right mouse button. In the *Watch* section it is possible to refer to a specific element of the *I* array or perform appropriate operations on it.

   In all cases a 2D array is displayed which is a slice of the 3D array, however by default this slice is in the wrong axis – a single line in 3 components is displayed. To get the whole image displayed for one component, change the axis.

- VSCode – under *SLICING* section, select the appropriate axis – set to 2 or select the appropriate indexing and press the *Apply* button.

Other components are available:

- VSCode – *Index* field.

8. Access to image parameters can be useful at work:

```
print(I.shape)  # dimensions /rows, columns, depth/
print(I.size)   # number of bytes
print(I.dtype)  # data type
```

The print function is a display to the console.

■

### 1.3.2 Loading, displaying and saving an image using the *Matplotlib* module

An alternative way to implement input/output operations is to use the *Matplotlib* library. Then the handling of load/display etc. is similar to that known from the Matlab package. Documentation of the library is available online Matplotlib.

**Exercise 1.2** Do a task in which you practice handling files using the Matplotlib library. To keep some order in your code, create a new source code file.

1. Load *pyplot* module.

```
import matplotlib.pyplot as plt
```

(as allows you to shorten the name to be used in the project)

2. Load the same image *mandril.jpg*

```
I = plt.imread('mandril.jpg')
```

3. The display is implemented very similarly to the Matlab package:

```
plt.figure(1)         # create figure
plt.imshow(I)         # add image
plt.title('Mandril')  # add title
plt.axis('off')       # disable display of the coordinate system
plt.show()            # display
```

4. Image storage:

```
plt.imsave('mandril.png',I)
```

5. During the lab, it can also be useful to display certain elements in the image – such as points or frames.
6. Adding the display of points:

```
x = [ 100, 150, 200, 250]
y = [ 50, 100, 150, 200]
plt.plot(x,y,'r.',markersize=10)
```

Commas are necessary (unlike in Matlab) when setting array values. In the plot command, the syntax is similar to Matlab – 'r' – colour, '.' – dot, and the size of the marker. Full list of possibilities in the documentation.

7. Adding rectangle display – drawing shapes i.e. rectangles, ellipses, circles etc. is available

in *matplotlib.patches*. Please note the comments in the code below.

```
from matplotlib.patches import Rectangle # add at the top of the file

fig,ax = plt.subplots(1) # instead of plt.figure(1)

rect = Rectangle((50,50),50,100,fill=False, ec='r'); # ec - edge colour
ax.add_patch(rect) # display
plt.show()
```

■

## 1.4 Colour space conversion

### 1.4.1 OpenCV

The function *cvtColor* is used to convert the colour space.

```
IG = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)
IHSV = cv2.cvtColor(I, cv2.COLOR_BGR2HSV)
```

R Please note that in OpenCV the reading is in **BGR** order, not RGB. This can be important when manually manipulating pixels. For a full list of available conversions with the relevant formulas in OpenCV documentation

**Exercise 1.3** Convert the colour space of the given image.

1. Convert the *mandril.jpg* image to grayscale and HSV space. Display the result.
2. Display the H, S, V components of the image after conversion.

■

R Please note that in contrast to e.g. the Matlab package, here the indexing is from 0.

Useful syntax:

```
IH = IHSV[:,:,0]
IS = IHSV[:,:,1]
IV = IHSV[:,:,2]
```

### 1.4.2 Matplotlib

Here the choice of available conversions is quite limited. Therefore, we need to use the formulas for colour space conversion.

1. RGB to greyscale.

$$G = 0.299 \cdot R + 0.587 \cdot G + 0.144 \cdot B \tag{1.1}$$

The formula can be represented as a function:

```
def rgb2gray(I):
  return 0.299*I[:,:,0] + 0.587*I[:,:,1] + 0.114*I[:,:,2]
```

R The colour map must be set when displaying. Otherwise the image will be displayed in the default, which is not greyscale: `plt.gray()`

2. RGB do HSV.

```
import matplotlib # add at the top of the file
I_HSV = matplotlib.colors.rgb_to_hsv(I)
```

## 1.5 Scaling, rescaling with OpenCV

**Exercise 1.4** Scale the image with *mandrill*. Use the *resize* function to scale. ■

Example of use:

```
height, width =I.shape[:2] # retrieving elements 1 and 2, i.e. the corresponding
    height and width
scale = 1.75   # scale factor
Ix2 = cv2.resize(I,(int(scale*height),int(scale*width)))
cv2.imshow("Big Mandrill",Ix2)
```

## 1.6 Arithmetic operations: addition, subtraction, multiplication, difference module

The images are matrices, so the arithmetic operations are quite simple – just like in the Matlab package. Of course, remember to convert to the appropriate data type. Usually double format will be a good choice.

**Exercise 1.5** Perform arithmetic operations on the image *lena*.

1. Download the image *lena* from the course page, then load it using a function from OpenCV – add this code snippet to the file that contains the image loader *mandril*. Perform the conversion to grayscale. Add the matrices containing Mandril and Leny in grayscale. Display the result.
2. Similarly perform the subtraction and multiplication of images.
3. Implement a linear combination of images.
4. An important operation is the module from image difference. It can be done manually – conversion to the appropriate type, subtraction, module (`abs`), conversion to *uint8*. An alternative is to use the function *absdiff* from OpenCV. Please calculate the modulus from the difference of the mandril and Lena greyscale image.

■

R To display the image correctly, the data type must be changed to uint type. Appropriate conversion:

```
import numpy as np
cv2.imshow("C",np.uint8(C))
```

## 1.7 Histogram calculation

Calculating the histogram can be done using the function  textcalcHist. But before we get to that, let's remind ourselves of the basic control structures in Python – functions and subroutines. Please complete the following function yourself, which calculates a histogram from an image with 256 grayscale values:

```
def hist(img):
  h=np.zeros((256,1), np.float32) # creates and zeros single-column arrays
  height, width =img.shape[:2] # shape - we take the first 2 values
    for y in range(height):
...
return h
```

In Python, indentation is important as it determines the block of a function, or loop!

The histogram can be displayed using the function `plt.hist` or `plt.plot` from the *Matplotlib* library.

The function `calcHist` can count the histogram of several images (or components). However, the form of the formula is most commonly used:

```
hist = cv2.calcHist([IG],[0],None,[256],[0,256])
# [IG] -- input image
# [0] -- for greyscale images there is only one channel
# None -- mask (you can count the histogram of a selected part of the image)
# [256] -- number of histogram bins
# [0 256 ] -- the range over which the histogram is calculated
```

Please check that the histograms obtained by both methods are the same.

## 1.8 Histogram equalisation

Histogram equalisation is a popular and important pre-processing operation.

1. Classical histogram equalization is a global method, it is performed on the whole image. There is a ready-to-use function for this type of equalization in the OpenCV library:

```
IGE = cv2.equalizeHist(IG)
```

2. CLAHE Histogram Equalization (*Contrast Limited Adaptive Histogram Equalization*) – is an adaptive method that improves lighting conditions in an image by using histogram equalization for individual parts of the image rather than the whole image. The method works as follows:
   - division of the image into disjoint (square) blocks,
   - calculation of the histogram in blocks,
   - performing a histogram equalisation, where the maximum height of the histogram is limited and the excess is redistributed to adjacent intervals,
   - interpolation of pixel values based on calculated histograms for given blocks (four neighbouring quadrant centres are taken into account).

   For details, see Wiki and tutorial.

```
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
# clipLimit - maximum height of the histogram bar - values above are distributed
    among neighbours
# tileGridSize - size of a single image block (local method, operates on separate
    image blocks)
I_CLAHE = clahe.apply(IG)
```

**Exercise 1.6** Run and compare both equalisation methods. ■

## 1.9 Filtration

Filtering is a very important group of operations on images. As an exercise, please run:

- Gaussian filtration (`GaussianBlur`)
- Sobel filtration (`Sobel`)
- Laplasian filter (`Laplacian`)
- Median filtration (`medianBlur`)

R OpenCV documentation – filtration.

Please also note the other functions available:

- bilateral filtration,
- Gabor filters,
- morphological operations.

**Exercise 1.7** Run and compare both equalisation methods. ■