Wrocław University of Science and Technology

Artificial Intelligence and Computer vision laboratory classes

Lab 5 – Filtering

Kacper Haręzga 249111

[Repository Link]

1. Introduction

During this labs we got familiar with the different type of filters such as Gaussian, Median or Laplacian filter. Also, we learned a bit about different types of noises such as Salt and Pepper noise or gaussian noise.

2. Code description

a. Used libraries

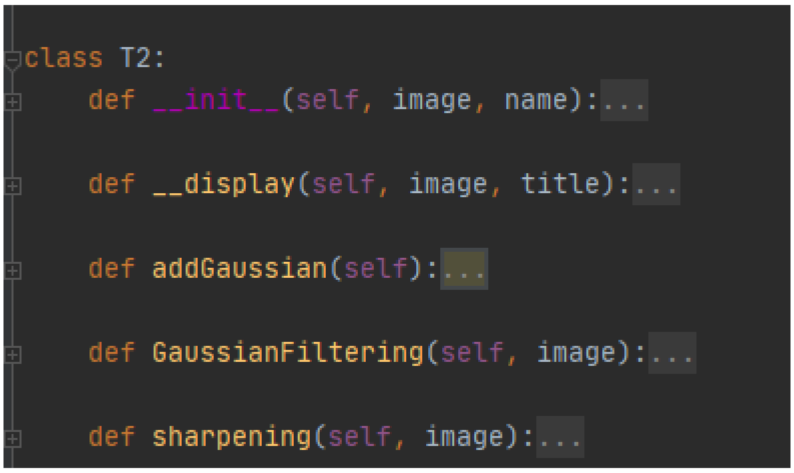
In the beginning of the provided source code naturally one can distinguish two essential libraries needed for the our realization. Short description of each lib:

- **Random** – used in Salt and Pepper noise implementation.
- **OpenCV** the most crucial import. Needed for the resizing, interpolating and histogram calculation.

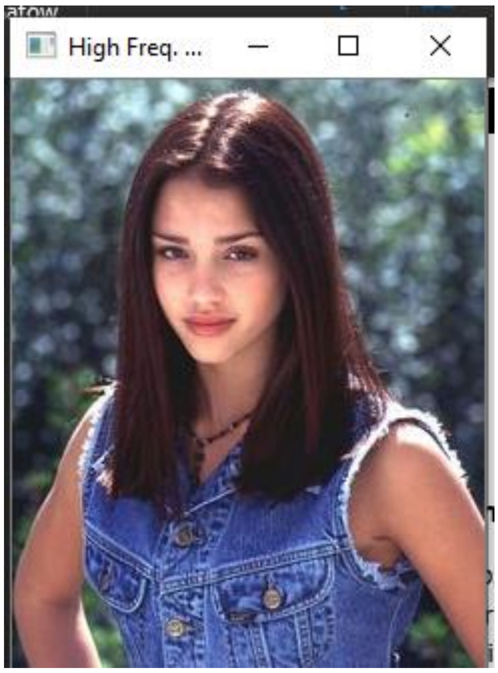
b. Tasks implementation

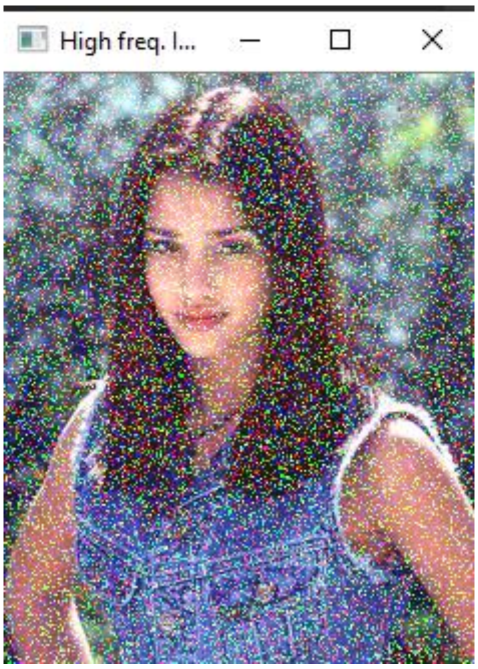
- **Task 1**

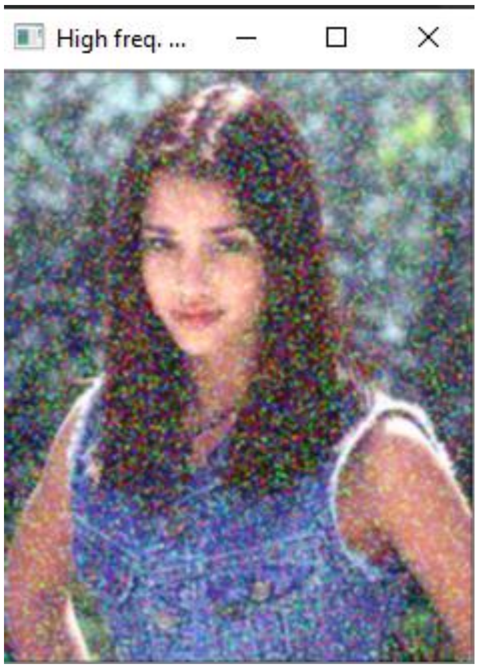
The goal of the first task was to play a little with the Gaussian noises and filtering. It's worth pointing out that Gaussian noise is kind of noise which can take only values that are Gaussian-distributed. In order to perform such operations I have designed class T2. When it comes to the relationship between the frequency of an image and the effect of noise and the filter on it, it is easy to learn that a high-resolution image can handle processing better. On the low frequency image some 'scars' or artefact are easily visible while high freq. one seems to be almost unchanged. The images are shown below are as following original->noised->filtered ->sharpened

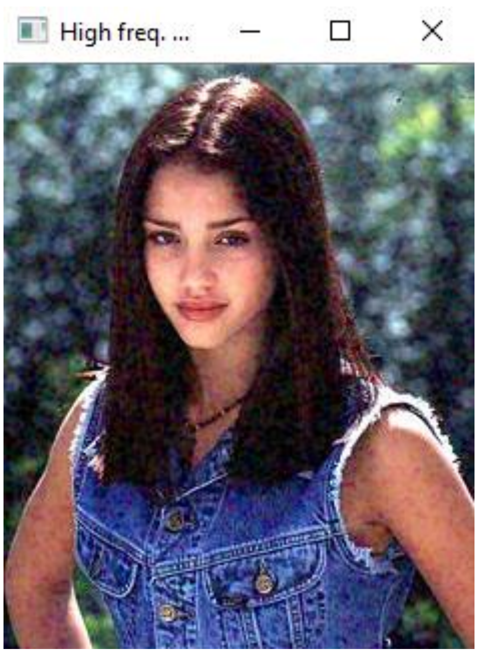


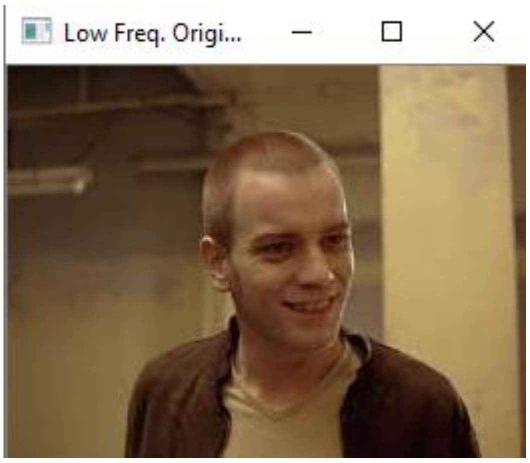
```
class T2:
    def __init__(self, image, name):...

    def __display(self, image, title):...

    def addGaussian(self):...

    def GaussianFiltering(self, image):...

    def sharpening(self, image):...
```

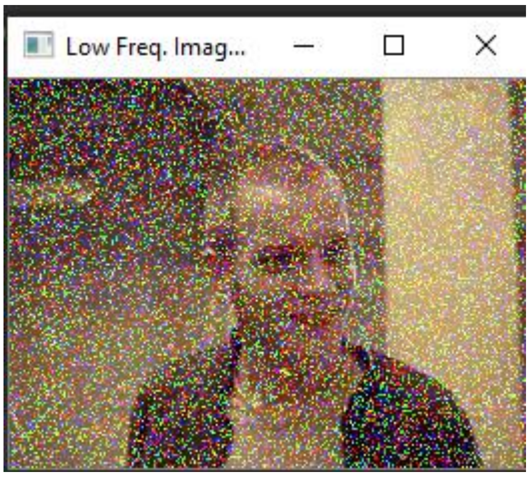

High Freq. ...

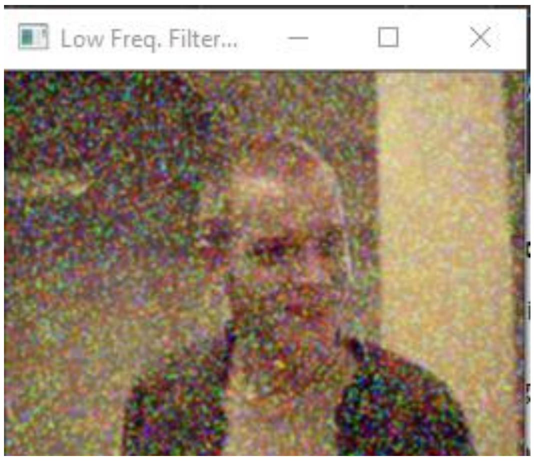

High freq. I...

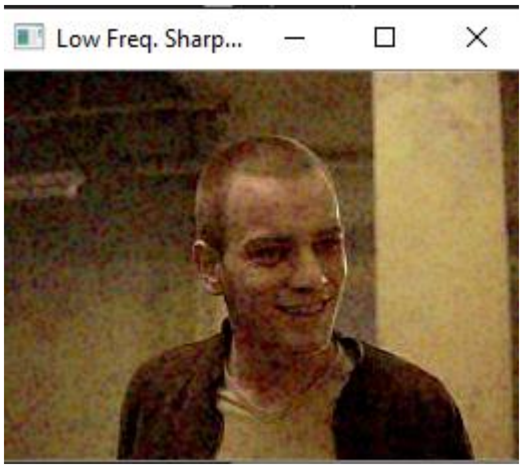

High freq. ...


High freq. ...



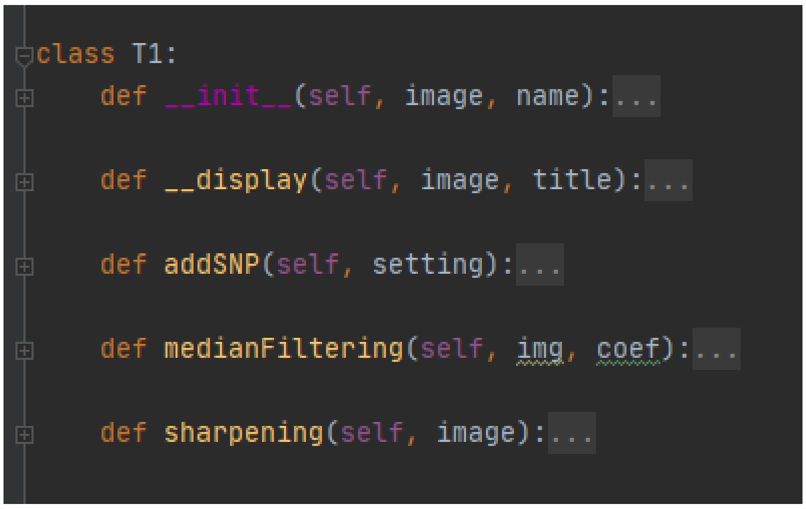




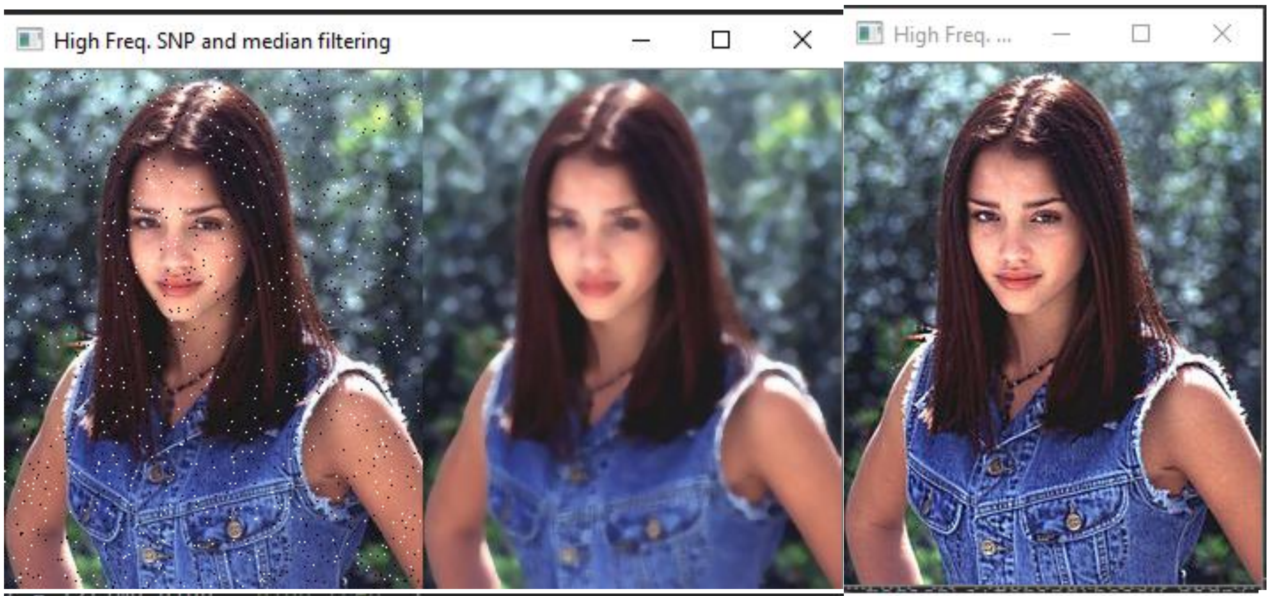


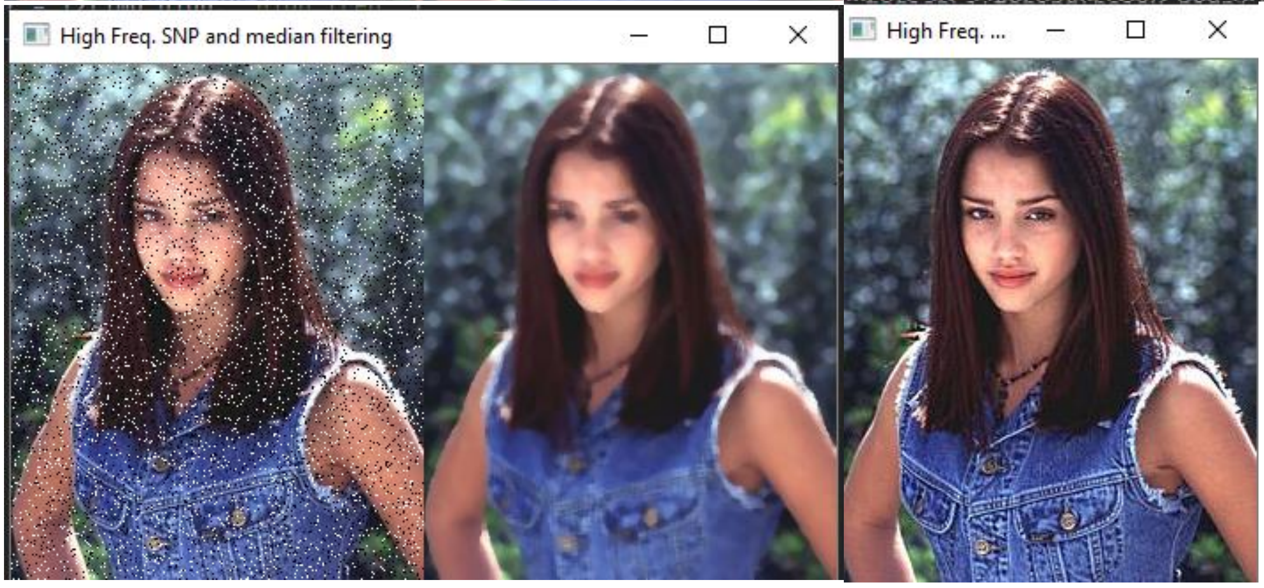
- **Task 2**

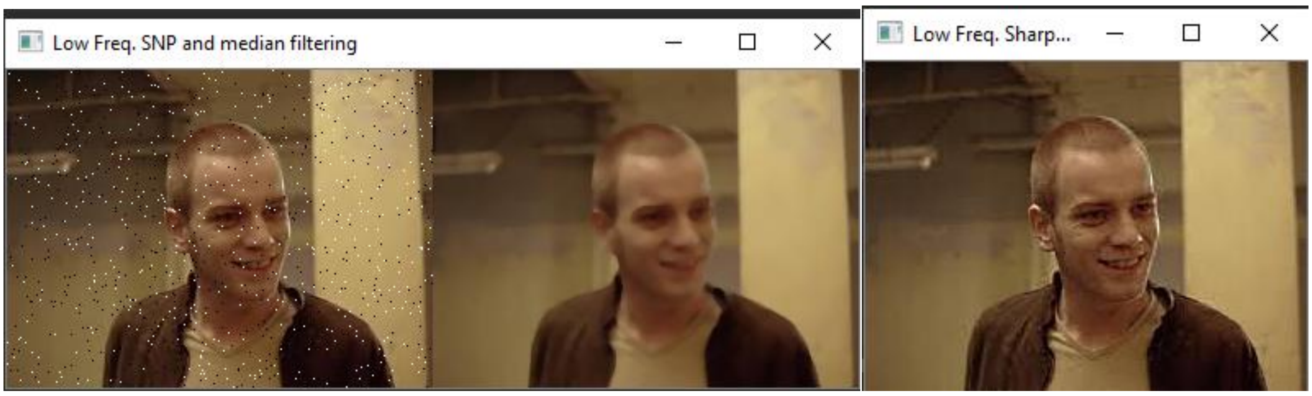
This task was very similar to the first one but this time we used SNP noise and median filtering. Class T1 is responsible for the realization of these problems. The median filter performs very well (tested on diff. coef) and allows us to repair even highly noised image. It take the neighborhood pixels and takes the median of them in order to filter image. I don't see any differences between HF and LF image. That might be result of the bad image selection.



```
class T1:
    def __init__(self, image, name):...

    def __display(self, image, title):...

    def addSNP(self, setting):...

    def medianFiltering(self, img, coef):...

    def sharpening(self, image):...
```

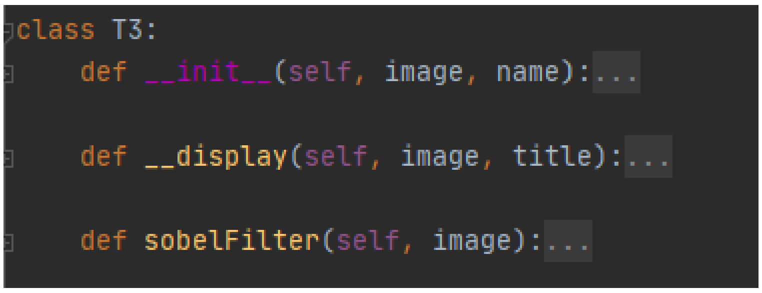

High Freq. SNP and median filtering
High Freq. ...


High Freq. SNP and median filtering
High Freq. ...

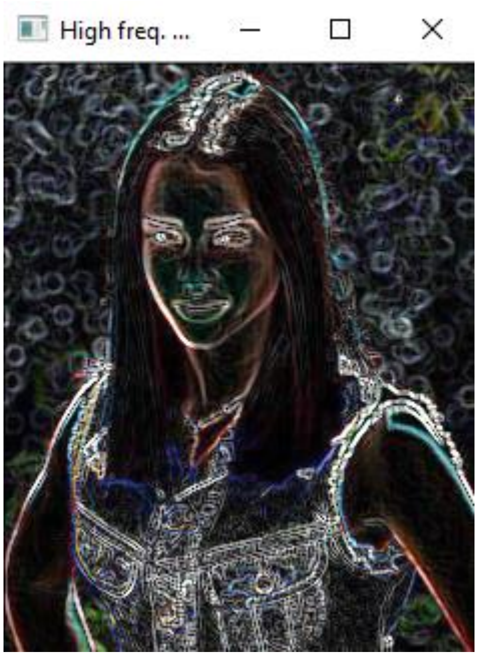

Low Freq. SNP and median filtering
Low Freq. Sharp...

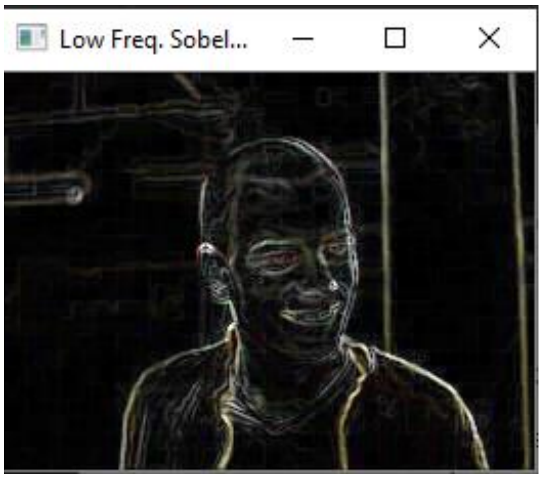
- **Task 3**

The Sobel filter was implemented using the T3 class. This filter is used to detect edges in an image by approximating the directional derivatives of the image intensities in eight directions. The filter itself seems to work better for lower frequency images. This may be due to the lower number of approximations in the case of a low-frequency image.



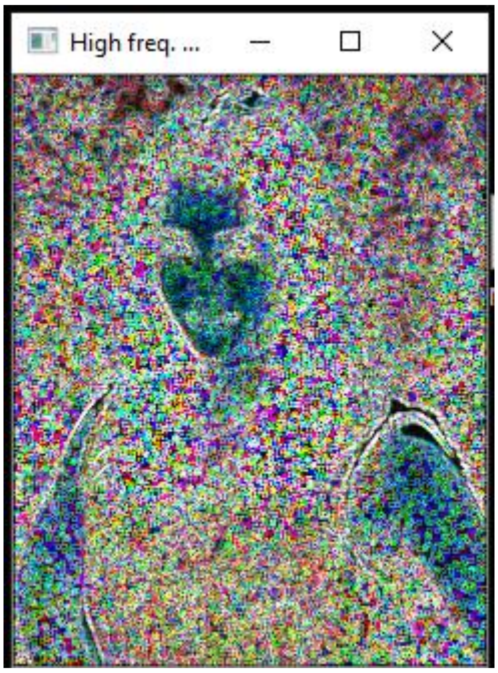
```
class T3:
    def __init__(self, image, name):...

    def __display(self, image, title):...

    def sobelFilter(self, image):...
```





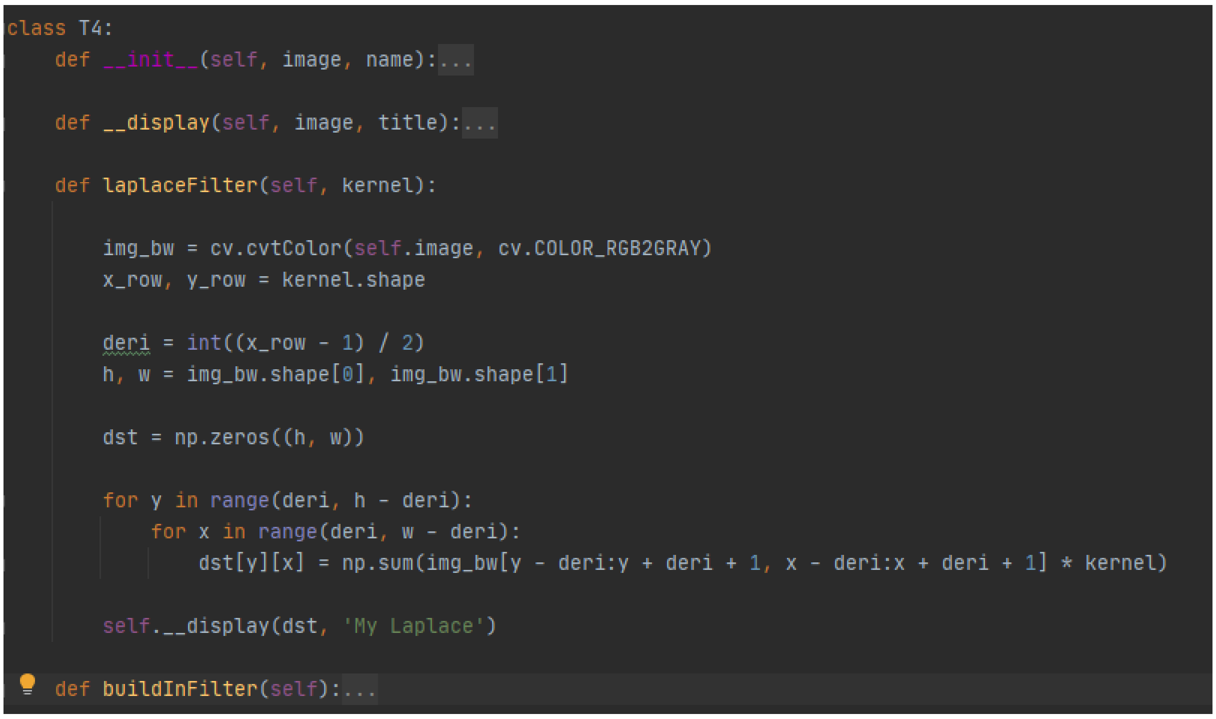
- **Task 4**

The aim of task 4 was to filter an image noisy with Gaussian noise with the Sobel filter. As you can see, this filter does not work at all for such images due to its operating principle. Gaussian noise completely distorts the gradient on which the Sobel filter works.

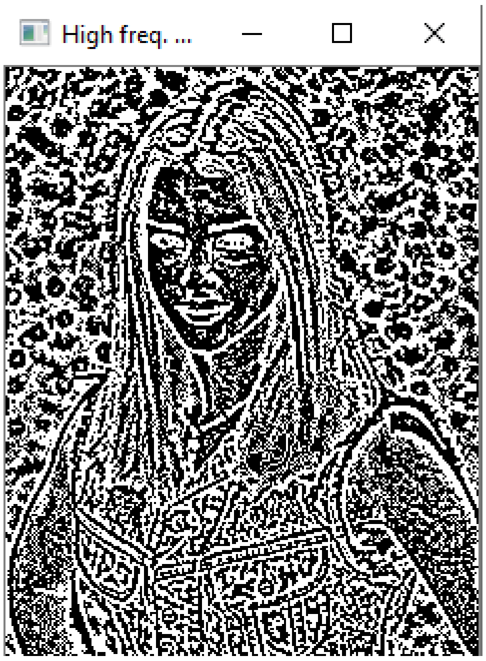


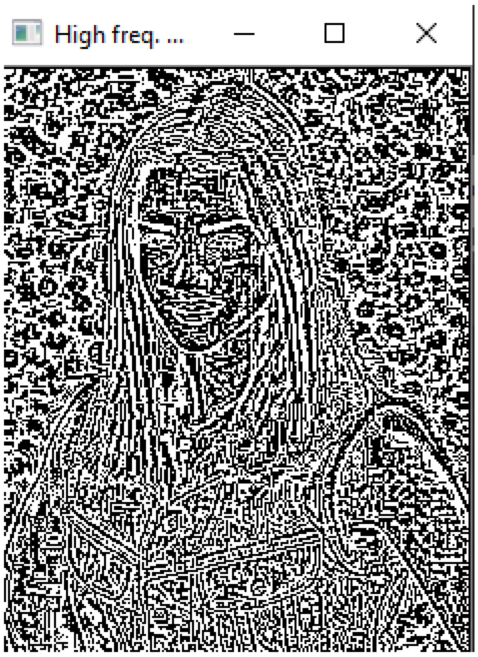
- **Task 5**

The task was to perform edge detection only this time using the Laplace filter. The goal was achieved using the built-in function and own implementation of the method that allows filtering with a given kernel.



```
class T4:
    def __init__(self, image, name):...

    def __display(self, image, title):...

    def laplaceFilter(self, kernel):

        img_bw = cv.cvtColor(self.image, cv.COLOR_RGB2GRAY)
        x_row, y_row = kernel.shape

        deri = int((x_row - 1) / 2)
        h, w = img_bw.shape[0], img_bw.shape[1]

        dst = np.zeros((h, w))

        for y in range(deri, h - deri):
            for x in range(deri, w - deri):
                dst[y][x] = np.sum(img_bw[y - deri:y + deri + 1, x - deri:x + deri + 1] * kernel)

        self.__display(dst, 'My Laplace')

    def buildInFilter(self):...
```





The Laplacian filter is an edge detector used to compute the second derivatives of an image, measuring the rate at which the first derivatives change. My implementation worked exactly like that. On the left is the result of the built-in function and on the right is the result of my method. Honestly, the built-in function seems to be more accurate, but the difference is small, which means that the implementation was correct. In my opinion the Laplace filter is much more effective than the Laplace filter.