Biometrics Project 01 Report

Igor Rudolf

Summer Semester 2025

1 Introduction

During the Biometrics summer semester 2025, I was tasked with designing and developing an image processing application. This report describes the application with theory needed to understand the implementation process. The report outlines analysis for the creation of the program as well as includes examples of pre and post processing images. Finally, the paper presents an analysis of the images processed with the help of diagrams.

2 Brief Description of the Application and GUI Aspects

2.1 What application does:

The application allows the user to open image and process it with individual needs. The user is able to do such basic operations like making image negative, turning it into shades of grey or use graphics filters such as sharpening or averaging. After that the end user is able to save the processed jpg file on local drive.

2.2 The technology used:

The whole application has been created using the tkinter python API. With the use of additional libraries such as: PIL, math, numpy and matplotlib. Below one's may read the reason why each technology has been used.

- **matplotlib:** Mainly the library was used for implementing the histogram to application which allows user to compare the histograms of intensity for original and modified image, as well as to see the results of horizontal and vertical projection about which later will be said more.
- **numpy:** All the implementation was used purely was computation on matrices such as summing elements in the rows or columns in the matrices. As well as to make easy copy of an array.
- **math:** used for basics mathematical operations such as calculating the nth root of a number

- **PIL:** Mainly used for getting the Image object. To enable `Back` operation (undoing the operation), program stores in the stack the data about previous images. To do the operation application makes copy of an image, stores it and does an operation on copied image. Thanks to this reversing operations is almost immediate.
- **tkinter:** Used for creating the GUI. Below explaining the layout of GUI.

2.3 Graphical user interface:

The application may be divided into two main parts. The first before processing to reading the image. Here the users see empty space with big title of application. To process user must read the image first by clicking: File->Read Image and then choosing jpg image from disc drive. After that the user will see the second part of application after reading the image.

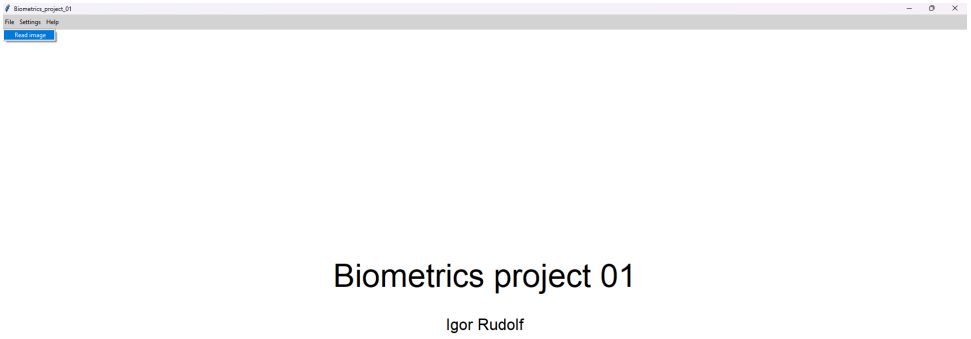

Figure 1: Welcome page, user sees it before reading the image.

The main part of the application can be divided into subregions just as the GUI is created.

- The `Top Bar` is placed on top of the application. It contains buttons such as "File", "Settings" and "Help". From which the "Help" section is yet not implemented. The "File" option allows the user to read again the image (also different one) and save the processed results. In "setting" user may choose different theme of application. Currently the LIGHT_THEME and DARK_THEME is implemented in the system.
- The 'Content' is divided into two main parts, the 'Left panel' and 'Right panel'.
  - The 'left panel' is a kind of a user command center. Here the user may perform many operations on image as well as to undo the operation. All the operations will be discussed in the further part of this report. Below the command center user may see the histograms of intensity for original as well as for the modified image.

– The 'right panel' is divided into two regions `top_subpanel` and `bottom_subpanel`. The first one shows image after several modifications and the bottom one the original one the one that we read.

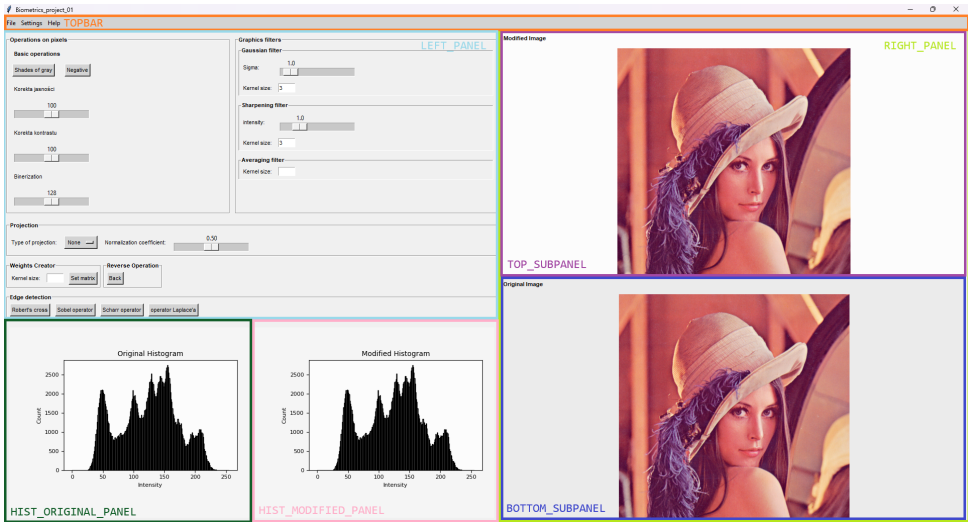

Figure 2: Main page, user sees it after reading the image.

3 Precise theoretical and practical aspects of the implemented methods

3.1 Description of application capabilities:

The image processing app have 4 major abilities such as doing basic opeartions on pixels, applying graphics filter, applying horizontal and vertical projection and an ability to detect edges.

For the basics opeartionn on pixel user may apply:

- **shades of gray (grayscale):** converses the image to grayscale.
- **negative:** creates the negative of the image (for each RGB channel: 255-x, x: the value for pixel).
- **correctness of brightness:** lights up or down the image from 0% to 200%.
- **correctness of contrast:** corrects the contrast from 0% to 200%.
- **binarization:** sets the pixels as black or white depending on threshold.

For the filter application can apply:

- **gaussian filter:** blurs the image with the use of gaussian kernel, the user may set the kernel size as well as sigma parameter.

- **sharpening filter:** edhances the edges in the image with the usage of kernel size and sigma parameter.
- **averaging filter:** simple blur, averages the values in some neighborhood.

For the projections we distinguish:

- **horizontal:** plot where every point is representation of sum of pixels in each row.
- **vertical:** plot where every point is representation of sum of pixels in each column.

the user may also set the coefficiant of normalization

For the Edge detection user can apply:

- **Robert's cross**
- **Sobel operator**
- **Scharr operator**
- **Laplace operator**

each methods detects edges, the difference is matrix needed for that.
Robert's cross requires matrix 2x2 from user wheras the rest requires matrix 3x3. In the application the user is required to set own weight matrix.

Apart from these major functionalities, the user may observe the changing histogram of brightness.
After each appearance the histogram of brightness for the modified image adapts to the operations performed. It is also possible to undo the operations performed previously.

3.2 Theoretical and implementation discussion of the presented methods

Below, each operation is described first from a theoretical point of view, and then followed by its pseudocode.

Operation on pixel which application covers may be described as:

3.2.1 Negative

Theory (unchanged):
To simply put, involves the inversion of pixel values. We can simply write this as: new_pixel_value = 255 - old_pixel_value. This operation is performed on every pixel.
Pseudocode:

Algorithm 1 Negative Image Operation

```
procedure NegativeImage(Image img)
    width ← img.width
    height ← img.height
    for y = 0 to height − 1 do
        for x = 0 to width − 1 do
            (r, g, b) ← img.getPixel(x, y)
            r ← 255 − r
            g ← 255 − g
            b ← 255 − b
            img.setPixel(x, y, (r, g, b))
        end for
    end for
    return img
end procedure
```

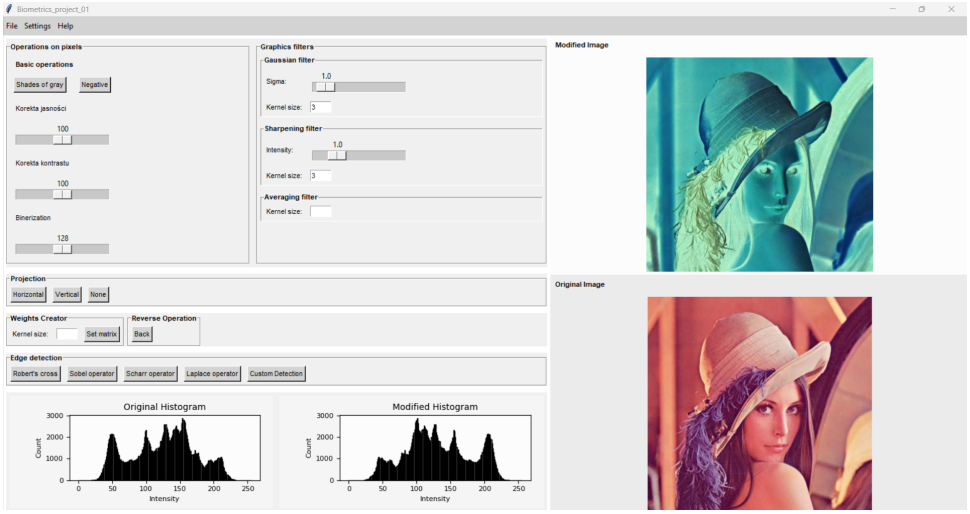

Figure 3: Above an example of application of the negative for: lena.png.

3.2.2 Grayscale

Theory (unchanged):
Converts the colorful image represented by the values (R, G, B) to the image with only one value per pixel, which is the intensity of brightness. So from (R, G, B) the user gets a single grayscale channel. The equation for a pixel looks like this: pixel_gray_value = $0.299 \times \mathrm{R} + 0.587 \times \mathrm{G} + 0.114 \times \mathrm{B}$. The pseudocode looks like this:

Pseudocode:

Algorithm 2 Convert Image to Grayscale

```
procedure ConvertToGrayscale(Image img)
    width ← img.width
    height ← img.height
    for y = 0 to height − 1 do
        for x = 0 to width − 1 do
            (R, G, B) ← img.getPixel(x, y)
            gray ← 0.299 × R + 0.587 × G + 0.114 × B
            img.setPixel(x, y, (gray, gray, gray))
        end for
    end for
    return img
end procedure
```

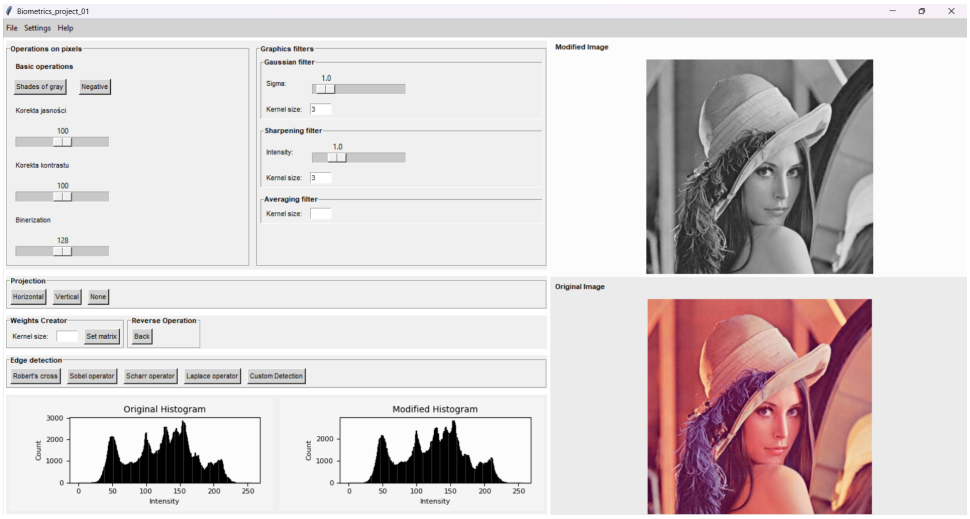

Figure 4: Above an example of application of the greyscale for: lena.png.

3.2.3 Brightness Correction

Theory (unchanged):
The brightening of image is operation about rescalling the values of each pixel in its (R, G, B) channel. The new value of each pixel is obtained after calculating: new_pixel_value = old_pixel_value * f, where f is the brightness correctness factor.

Pseudocode:

3.2.4 Contrast Correction

Theory (unchanged):
Contrast correction is about adjusting the difference between pixel values and the half scale of grey(128). With this operation user can increate the contrast of image. The equation looks like: new_pixel_value= 128 + f * (old_pixel_value -128). The intuition

Algorithm 3 Brightness Correction

procedure AdjustBrightness(Image img, Factor f)
    $w \leftarrow img$.width
    $h \leftarrow img$.height
    for $y = 0$ **to** $h - 1$ **do**
        for $x = 0$ **to** $w - 1$ **do**
            $(R, G, B) \leftarrow img$.getPixel(x, y)
            $R' \leftarrow \min(255, \max(0, R \times f))$
            $G' \leftarrow \min(255, \max(0, G \times f))$
            $B' \leftarrow \min(255, \max(0, B \times f))$
            img.setPixel$(x, y, (R', G', B'))$
        end for
    end for
    return img
end procedure

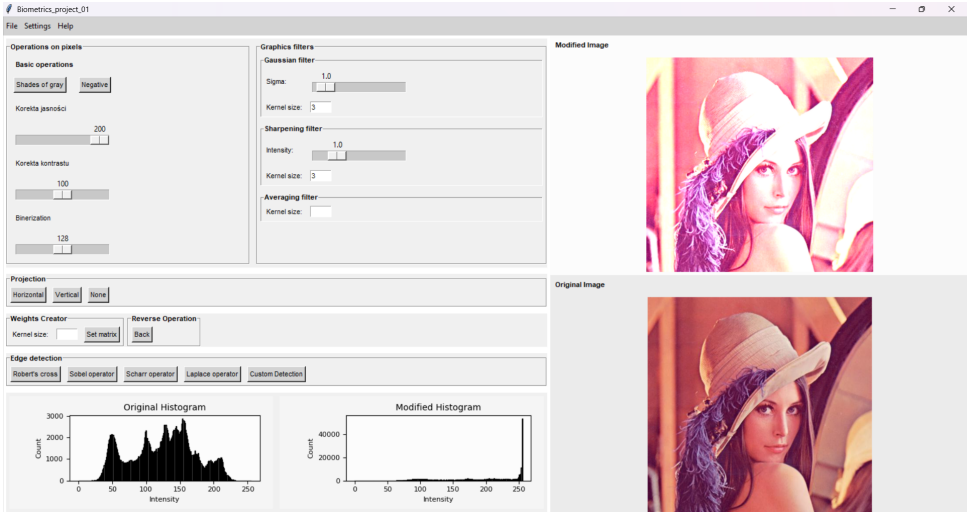

Figure 5: Above an example of application of the brightness correction for: lena.png.

with the f coeffician is that if f¿1 then brighter pixels become more brighter and dark pixels become more darker. If the f¡1 the effect is opposite, and if f=1, the there is no difference then from previous image.

Pseudocode:

Algorithm 4 Contrast Correction

procedure AdjustContrast(Image img, Factor f)
    $w \leftarrow img$.width
    $h \leftarrow img$.height
    for $y = 0$ **to** $h - 1$ **do**
        for $x = 0$ **to** $w - 1$ **do**
            $(R, G, B) \leftarrow img$.getPixel(x, y)
            $R' \leftarrow \min(255, \max(0, 128 + f \times (R - 128)))$
            $G' \leftarrow \min(255, \max(0, 128 + f \times (G - 128)))$
            $B' \leftarrow \min(255, \max(0, 128 + f \times (B - 128)))$
            img.setPixel$(x, y, (R', G', B'))$
        end for
    end for
    return img
end procedure

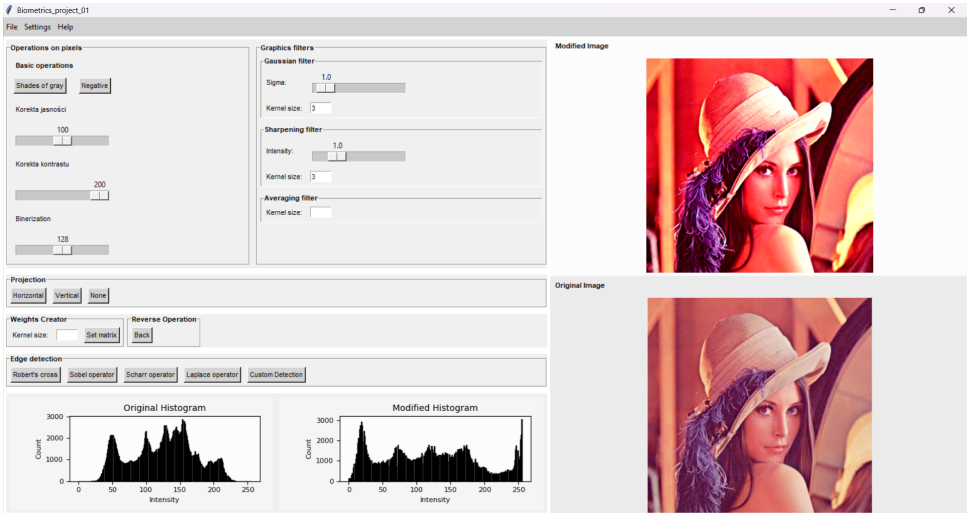

Figure 6: Above an example of application of the contrast correction for: lena.png.

3.2.5 Binarization

Theory (unchanged):
Binarization converts a grayscale image into binary image. Which means that every pixel value is assigned to one of two possible values. Either 0 or 255. If old_pixel_value ¿ threshold then pixel_value is 255, otherwise is 0. The user sets a threshold.

Pseudocode:

Algorithm 5 Binarization Operation

procedure BINARIZE(Image img, Threshold T)
    $w \leftarrow img$.width
    $h \leftarrow img$.height
    for $y = 0$ **to** $h - 1$ **do**
        for $x = 0$ **to** $w - 1$ **do**
            $(R, G, B) \leftarrow img$.getPixel(x, y)
            $gray \leftarrow 0.299 \times R + 0.587 \times G + 0.114 \times B$
            if $gray > T$ **then**
                $newPixel \leftarrow (255, 255, 255)$
            else
                $newPixel \leftarrow (0, 0, 0)$
            end if
            img.setPixel$(x, y, newPixel)$
        end for
    end for
    return img
end procedure

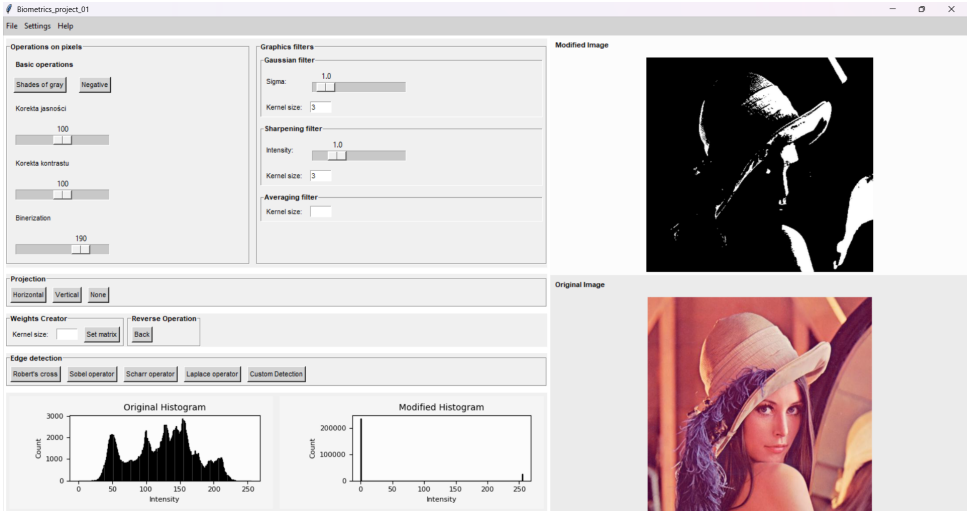

Figure 7: Above an example of application of the binarization for: lena.png.

Graphics Filters Implemented in the Application:

Gaussian Filter The Gaussian filter is a method whose main purpose is to eliminate noise from the image. We take into account the neighboring pixels, and the farther a pixel is from the center, the less influence it has. A key parameter of this function is the mask—essentially a matrix—whose values come from the normal distribution.

For every pixel, assuming we have a square mask (typically of odd size), we look at its neighborhood. A 3x3 kernel is a natural choice. Each pixel in the neighborhood is multiplied by the corresponding value from the kernel. These values are then summed, and the result becomes the new pixel value. In this way, we get a more blurred image.

In the application created for the biometrics course, the implementation allows the

user to control two parameters: `kernel size` and `sigma`. The intuition is that as `sigma` increases, the blur increases. This is because `sigma` corresponds to the width of the bell curve in the normal distribution—larger `sigma` means neighboring pixels (within the kernel size) have a more significant impact on the central pixel during the blur operation.

The formula for the Gaussian kernel is:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Pseudocode:

Algorithm 6 Gaussian Filter

procedure ApplyGaussianFilter(Image *img*, int *kernel_size*, float *sigma*)
    $width \leftarrow img$.width
    $height \leftarrow img$.height
    $result \leftarrow$ new image with same size as img
    $kernel \leftarrow$ GenerateGaussianKernel($kernel\_size$, $sigma$)
    $offset \leftarrow kernel\_size \div 2$
    for $y = 0$ to $height - 1$ **do**
        for $x = 0$ to $width - 1$ **do**
            $r\_acc \leftarrow 0$, $g\_acc \leftarrow 0$, $b\_acc \leftarrow 0$
            for $ky = -offset$ to $offset$ **do**
                for $kx = -offset$ to $offset$ **do**
                    $px \leftarrow \text{clamp}(x + kx, 0, width - 1)$
                    $py \leftarrow \text{clamp}(y + ky, 0, height - 1)$
                    $(r, g, b) \leftarrow img.getPixel(px, py)$
                    $weight \leftarrow kernel[ky + offset][kx + offset]$
                    $r\_acc \leftarrow r\_acc + r \cdot weight$
                    $g\_acc \leftarrow g\_acc + g \cdot weight$
                    $b\_acc \leftarrow b\_acc + b \cdot weight$
                end for
            end for
            $result.setPixel(x, y, (int(r\_acc), int(g\_acc), int(b\_acc)))$
        end for
    end for
    return $result$
end procedure

`kernel_of_the_gauss` creates a 2D list and computes values from the Gaussian distribution using the equation given above. The `apply_gaussian_filter` then goes through each pixel, processes its neighborhood, multiplies each neighbor by the kernel value, sums RGB values, and assigns the average as the new pixel value.

Algorithm 7 Generate Gaussian Kernel

procedure GENERATEGAUSSIANKERNEL(int *size*, float *sigma*)
    $kernel \leftarrow$ 2D array of size $size \times size$
    $center \leftarrow size \div 2$
    $sum \leftarrow 0$
    for $i = 0$ to $size - 1$ **do**
        for $j = 0$ to $size - 1$ **do**
            $x \leftarrow i - center$
            $y \leftarrow j - center$
            $kernel[i][j] \leftarrow e^{-\frac{x^2+y^2}{2 \cdot \sigma^2}}$
            $sum \leftarrow sum + kernel[i][j]$
        end for
    end for
    for $i = 0$ to $size - 1$ **do**
        for $j = 0$ to $size - 1$ **do**
            $kernel[i][j] \leftarrow kernel[i][j] \div sum$
        end for
    end for
    return *kernel*
end procedure

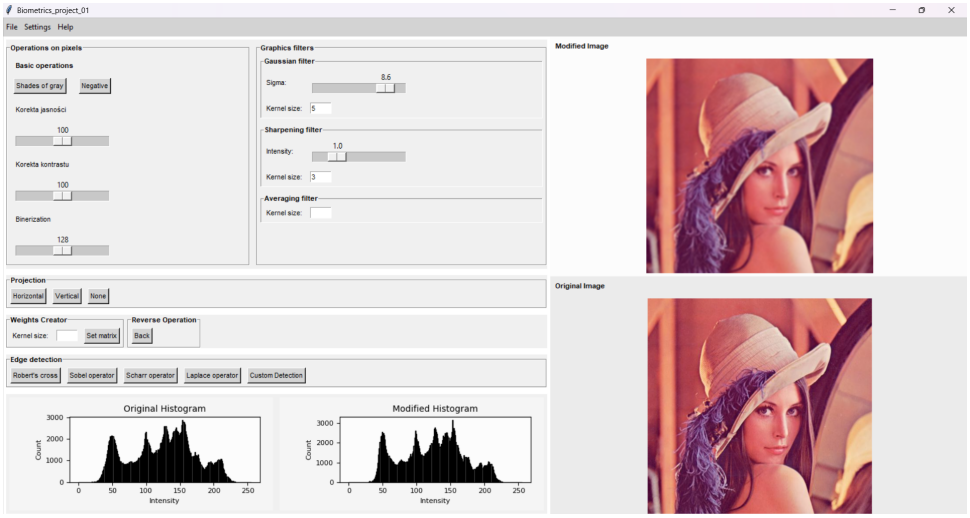

Figure 8: Above an example of application of the gaussian filter for: lena.png.

Sharpening Filter As the name suggests, the sharpening filter highlights the edges and subtle details in our image. Intuitively, it works opposite to the Gaussian filter, and one could say it is its inverse. In my implementation, it takes two parameters: `kernel_size` and `intensity`. The `kernel_size` is interpreted the same way as in the Gaussian filter. As for `intensity`, although its meaning is not the same as `sigma` in the Gaussian filter, we can say that a higher `intensity` leads to more pronounced details and a stronger sharpening effect.

The sharpening operation can be expressed as:

$$I_{sharpened} = I_{original} + \alpha(I_{original} - I_{blurred})$$

Here, α is the intensity controlled by the user. $I_{blurred}$ is the image after slight blurring using a Gaussian filter. The subtraction operation literally means that for each R, G, B channel we subtract the corresponding blurred value from the original.

Pseudocode:

Algorithm 8 Generate Sharpening Kernel

procedure SHARPENINGKERNEL(kernel_size, intensity)
    $total \leftarrow kernel\_size \times kernel\_size$
    $avg \leftarrow 1.0/total$
    $kernel \leftarrow$ matrix of size $kernel\_size \times kernel\_size$ with all values $-intensity \cdot avg$
    $center \leftarrow kernel\_size//2$
    $kernel[center][center] \leftarrow 1 + intensity - intensity \cdot avg$
    return $kernel$
end procedure

Algorithm 9 Apply Sharpening Filter

procedure APPLYSHARPENING(image, kernel_size, intensity)
    $kernel \leftarrow$ SharpeningKernel$(kernel_size, intensity)$
    $offset \leftarrow kernel\_size//2$
    for each pixel (x, y) in $image$ **do**
        $sum\_r, sum\_g, sum\_b \leftarrow 0$
        for $j = -offset$ to $offset$ **do**
            for $i = -offset$ to $offset$ **do**
                $(r, g, b) \leftarrow$ neighbor pixel at $(x + i, y + j)$
                $weight \leftarrow kernel[j + offset][i + offset]$
                $sum\_r += r \cdot weight$
                $sum\_g += g \cdot weight$
                $sum\_b += b \cdot weight$
            end for
        end for
        Set pixel (x, y) to $(clamp(sum\_r), clamp(sum\_g), clamp(sum\_b))$
    end for
    return $image$
end procedure

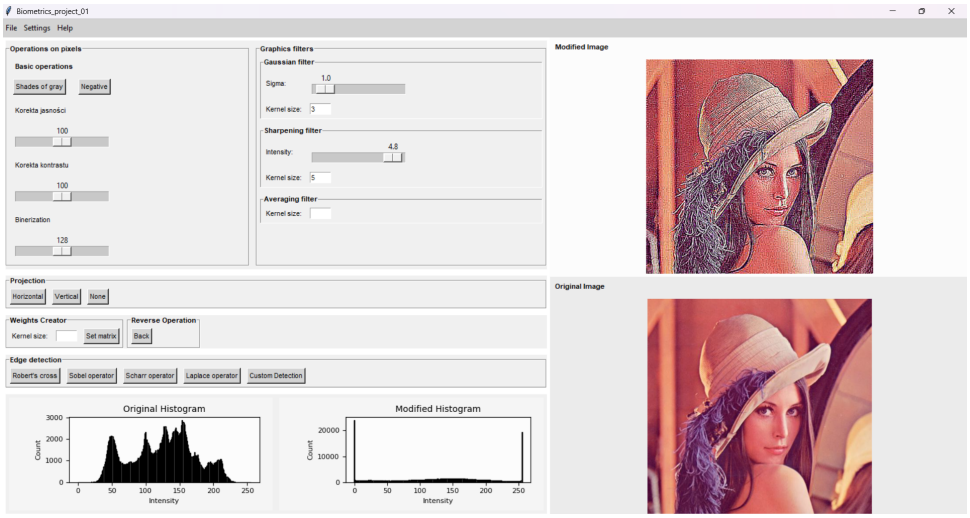

Figure 9: Above an example of application of the sharpening filter for: lena.png.

`sharpening_kernel`, just like `gaussian_kernel`, is responsible for creating the kernel. We set its center as $1 + intensity - intensity \cdot avg$. This can be interpreted as extracting subtle information from the image. The function `apply_sharpening_filter` simply goes through every pixel and applies the kernel.

Averaging Filter The averaging filter works by defining the `kernel_size` and then iteratively going through every pixel. For each pixel, we compute the value based on its defined neighborhood of size `kernel_size`. This algorithm smoothens the image, which is intuitive given the name `averaging`.

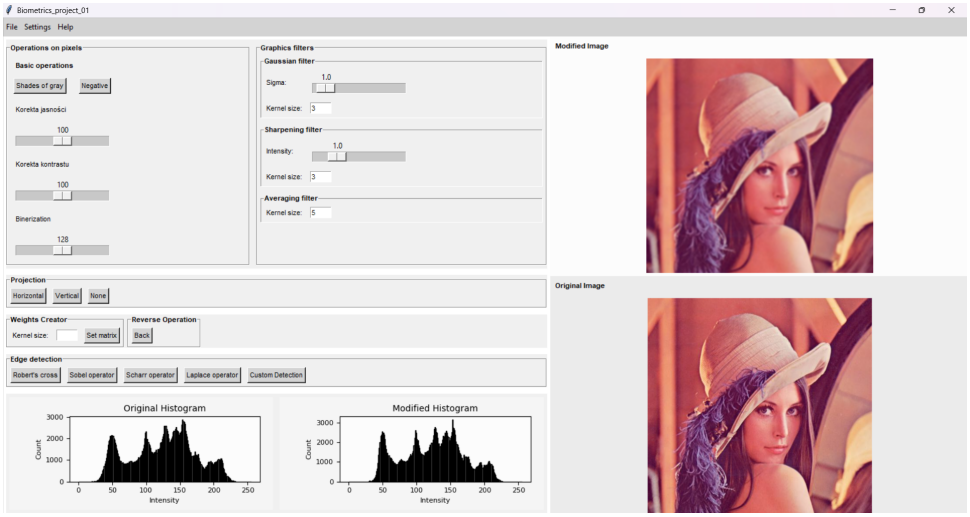

Figure 10: Above an example of application of the averaging filter for: lena.png.

Edge Detection Algorithms:

Roberts Cross Edge Detection

Roberts cross is a simple edge detection algorithm that identifies edges in an image by detecting pixel differences in a very small 2x2 neighborhood along the matrix diagonals. The main advantage of this algorithm is its speed.

The algorithm works by iteratively processing each pixel, determining its 2x2 neighborhood matrix, and then multiplying this matrix by a user-defined mask. The resulting value becomes the new pixel value. For RGB images, this process is applied to each color channel, thus determining new R, G, and B values.

The edge detection intuition is straightforward: edges are detected along two diagonals - from the top-left to bottom-right corner, and from the top-right to bottom-left corner.

Algorithm 10 Roberts Cross Edge Detection (Standard)

procedure RobertsCross(Image img)
    Convert img to grayscale
    $width \leftarrow$ image width
    $height \leftarrow$ image height
    $new\_img \leftarrow$ new grayscale image of size $(width, height)$
    for $y = 0$ to $height - 2$ **do**
        for $x = 0$ to $width - 2$ **do**
            $gx \leftarrow src[x, y] - src[x + 1, y + 1]$
            $gy \leftarrow src[x + 1, y] - src[x, y + 1]$
            $gradient \leftarrow \sqrt{gx^2 + gy^2}$
            $new\_img[x, y] \leftarrow \min(255, \lfloor gradient \rfloor)$
        end for
    end for
    for $y = 0$ to $height - 1$ **do**
        $new\_img[width - 1, y] \leftarrow 0$
    end for
    for $x = 0$ to $width - 1$ **do**
        $new\_img[x, height - 1] \leftarrow 0$
    end for
    return new_img
end procedure

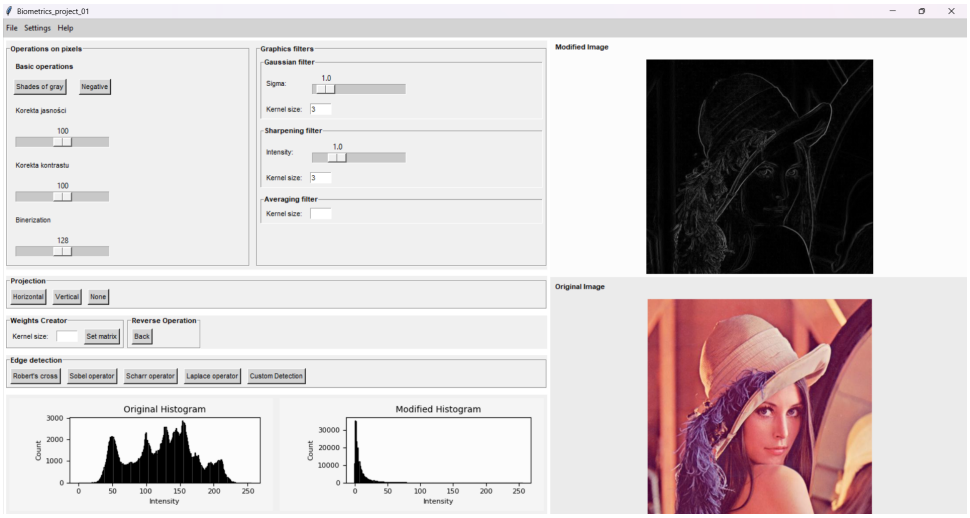

Figure 11: Above an example of application of the roberts cross for: lena.png.

3.3 Sobel Operator

The key difference with the Sobel operator is the use of a 3x3 matrix. Typically, the algorithm uses two 3x3 matrices that can be denoted as Gx and Gy. However, this implementation is simpler, using only one matrix to calculate a single component either vertically or horizontally.

This approach is motivated by creating a simple tool for users. The reasoning is similar to the Roberts cross method - multiplying the neighborhood matrix with a weight matrix/mask. For example, to detect horizontal edges (changes in the vertical direction), one might use the matrix:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

For vertical edge detection (changes in the horizontal direction), a similar approach is used:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Algorithm 11 Sobel Edge Detection

procedure SobelOperator(Image img)
    Convert img to grayscale
    $width \leftarrow$ image width
    $height \leftarrow$ image height
    $new\_img \leftarrow$ new grayscale image of size $(width, height)$
    Define G_x and G_y Sobel kernels
    for $y = 1$ to $height - 2$ **do**
        for $x = 1$ to $width - 2$ **do**
            $gx \leftarrow 0$, $gy \leftarrow 0$
            for $j = -1$ to 1 **do**
                for $i = -1$ to 1 **do**
                    $pixel \leftarrow src[x + i, y + j]$
                    $gx \leftarrow gx + G_x[j + 1][i + 1] \cdot pixel$
                    $gy \leftarrow gy + G_y[j + 1][i + 1] \cdot pixel$
                end for
            end for
            $g \leftarrow \sqrt{gx^2 + gy^2}$
            $new\_img[x, y] \leftarrow \min(255, \lfloor g \rfloor)$
        end for
    end for
    return new_img
end procedure

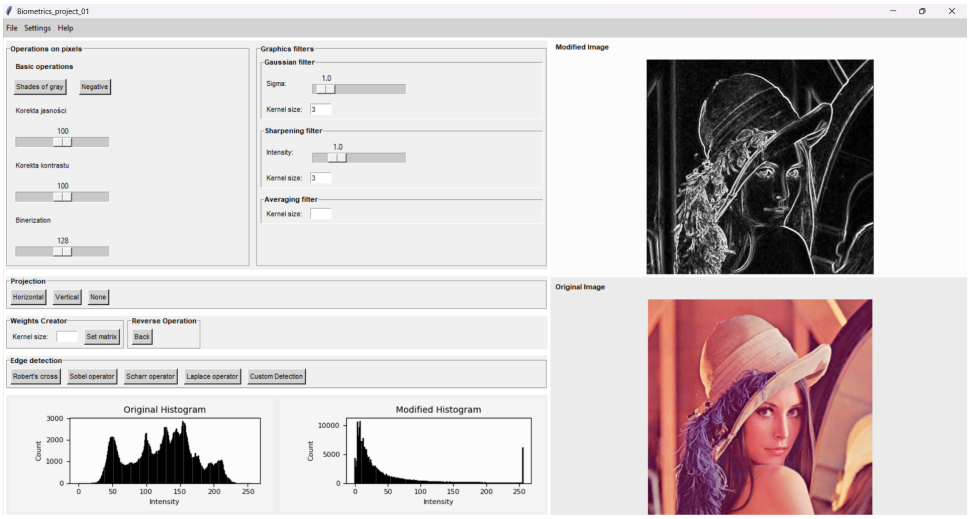

Figure 12: Above an example of application of the sobel operator for: lena.png.

3.4 Scharr Operator

The Scharr operator is another edge detection filter using a 3x3 matrix. A key difference between Scharr and Sobel is that Scharr performs slightly better in detecting edges, especially in images with many fine details.

The Scharr operator uses two predefined masks: one for detecting vertical edges and

another for horizontal edges. The vertical edge detection mask is:

$$\begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix}$$

And the horizontal edge detection mask is:

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

It's worth noting that the masks differ significantly in their side rows compared to the Sobel operator.

Algorithm 12 Scharr Edge Detection

```
1: procedure ScharrOperator(Image img)
2:     Convert img to grayscale
3:     width ← image width
4:     height ← image height
5:     new_img ← new grayscale image of size (width, height)
6:     Define G_x and G_y Scharr kernels
7:     for y = 1 to height − 2 do
8:         for x = 1 to width − 2 do
9:             gx ← 0, gy ← 0
10:            for j = −1 to 1 do
11:                for i = −1 to 1 do
12:                    pixel ← src[x + i, y + j]
13:                    gx ← gx + G_x[j + 1][i + 1] · pixel
14:                    gy ← gy + G_y[j + 1][i + 1] · pixel
15:                end for
16:            end for
17:            g ← √(gx² + gy²)
18:            new_img[x, y] ← min(255, ⌊g⌋)
19:        end for
20:    end for
21:    return new_img
22: end procedure
```

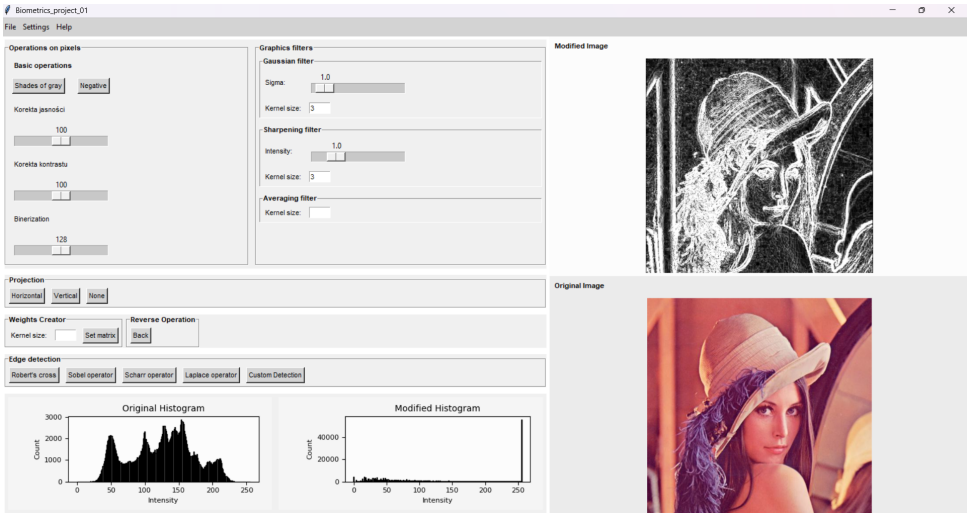

Figure 13: Above an example of application of the scharr operator for: lena.png.

3.5 Laplace Operator

The Laplace operator is another edge detection method, but with a crucial difference from Scharr and Sobel: it is non-directional. This means it detects brightness changes regardless of orientation.

The Laplace operator uses second-order derivatives. More precisely, it is the sum of second-order derivatives of image intensity along x and y coordinates. Mathematically, this can be expressed as:

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Here, f represents the image intensity function at a given point. In practice, this derivative is estimated using a mask:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The algorithm works by converting the image to grayscale, then for each pixel, taking its 3x3 neighborhood, multiplying the neighborhood values by the Laplace matrix, and truncating the result to the range [0, 255].

Algorithm 13 Apply Laplace Operator

```
1: procedure LaplaceOperator(Image img, Matrix weight_matrix)
2:     if weight_matrix is not 3 × 3 then
3:         raise error "Weights matrix must be 3x3"
4:     end if
5:     Convert img to grayscale
6:     w ← width of image
7:     h ← height of image
8:     new_img ← new grayscale image of size (w, h)
9:     for y = 1 to h − 2 do
10:        for x = 1 to w − 2 do
11:            acc ← 0
12:            for j = −1 to 1 do
13:                for i = −1 to 1 do
14:                    acc ← acc + src[x + i, y + j] · weight_matrix[j + 1][i + 1]
15:                end for
16:            end for
17:            acc ← |acc|
18:            if acc > 255 then
19:                acc ← 255
20:            end if
21:            dst[x, y] ← ⌊acc⌋
22:        end for
23:    end for
24:    return new_img
25: end procedure
```

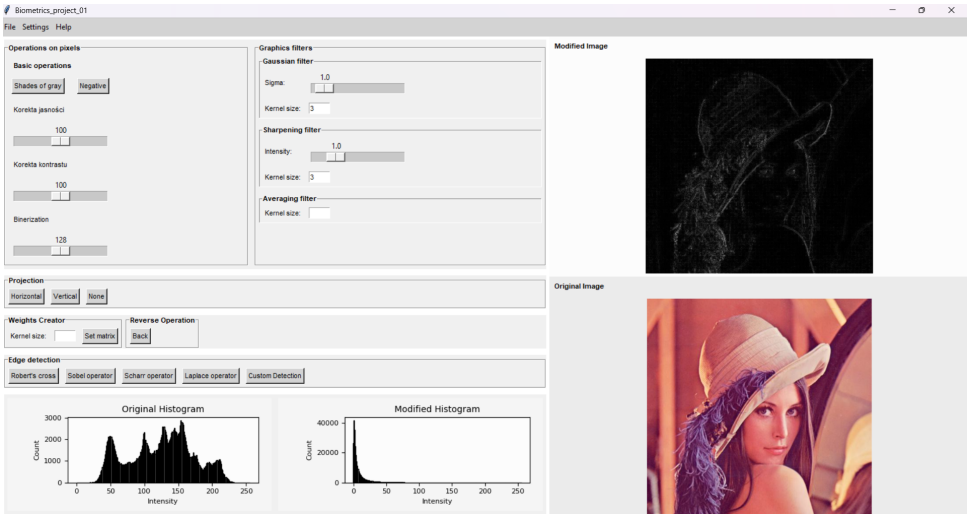

Figure 14: Above an example of application of the laplace operator for: lena.png.

3.6 Custom Edge Detection

The application also allows the creation of square matrices of any dimension (at least 2x2) for the purpose of creating any method of edge detection. The logic of this solution is relatively simple.

If the kernel_size provided by the user is of dimension 2x2, the algorithm works similarly to the `roberts_cross` method. The only difference is that in the `roberts_cross` algorithm, a specific matrix defines this algorithm, while when the user provides a custom matrix, the steps are performed as in the `roberts_cross` algorithm, but for the user-defined matrix.

If the user provides a matrix of at least 3x3 dimension, the Sobel algorithm is executed. However, it is not performed on the predefined matrix:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

as specified in the `sobel_operator`. Instead, the operations are performed as in the Sobel algorithm but with the matrix provided by the user.

Nevertheless, if the user provides a matrix in `custom_weights` that matches the matrix defined for `roberts_cross`, then the Custom Detection works identically to the `roberts_cross` algorithm. Similarly, if the user provides a matrix that matches the one used in the Sobel algorithm, the Sobel algorithm will be performed identically.

Let's now examine some sample weights used for testing the algorithm.

Figure 15: 1-st image: rotated variant of roberts cross, 2-nd image: vertical variant of prewitt operator, 3-td image: horizontal variant of prewitt operator.

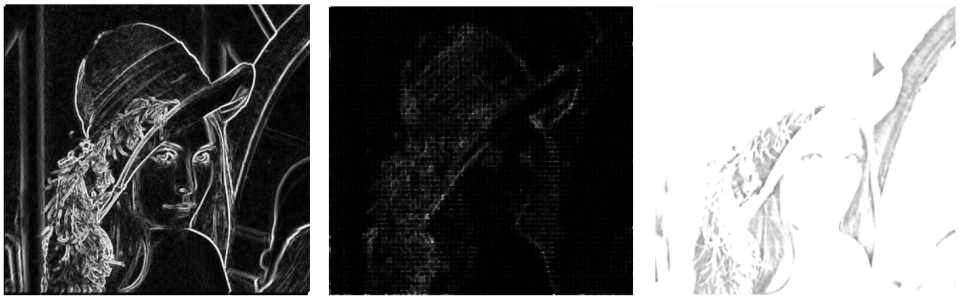

Figure 16: 1-st image: for matrix detecting 45 degree edges, 2-nd image: for matrix like modified Laplace operator, 3-td image: triangle like matrix(below more details)

For the above images (Figure 16) the matrices are as follows:

$$\text{First image: } \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} \quad \text{Second image: } \begin{bmatrix} -1 & 2 & -1 \\ 2 & -4 & 2 \\ -1 & 2 & -1 \end{bmatrix} \quad \text{Third image: } \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Overall, it seems like custom matrices can give interesting results as well.

Image Projection:

In general, image projection is a way of simplifying the representation of image content by summing pixel values along one of the dimensions. We refer here to the horizontal and vertical dimensions, meaning along rows or columns.

When it comes to horizontal projection, we sum the pixel values in each row. In this way, we obtain many values—exactly 'height' values of the image—which we then simply plot on a line chart. The X-axis simply indicates the row number, while the Y-axis corresponds to the brightness of the sum of pixels in the given row or column. Before moving to the pseudocode, it's worth considering the interpretation of such a plot. First and

foremost, high values on the plot indicate a row/column with bright areas. If a clearly visible fragment of the plot has noticeably high values, this also indicates a region with bright areas. The reasoning is identical in the case of low values and dark regions.

In the application, the following projections have been implemented:

- vertical
- horizontal

The algorithms are very similar— the only difference is that in one we iterate over columns and in the other over rows. Below I include the pseudocode for vertical and horizontal projection, which can be found implemented in my application.

Algorithm 14 Horizontal Projection

```
1: procedure HorizontalProjection(Image img)
2:     if img is not grayscale then
3:         convert img to grayscale
4:     end if
5:     array ← convert img to 2D pixel array
6:     height ← number of rows in array
7:     width ← number of columns in array
8:     projection ← new array of size height
9:     for y = 0 to height − 1 do
10:        sum ← 0
11:        for x = 0 to width − 1 do
12:            sum ← sum + array[y][x]
13:        end for
14:        projection[y] ← sum
15:    end for
16:    return projection
17: end procedure
```

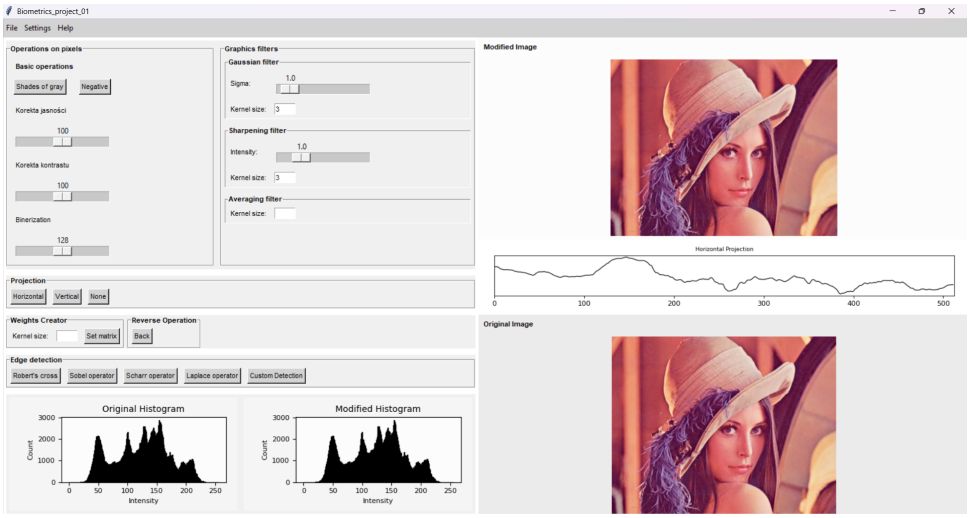

Figure 17: Above an example of application of the horizontal projection for: lena.png.

Algorithm 15 Vertical Projection

```
1:  procedure VerticalProjection(Image img)
2:      if img is not grayscale then
3:          convert img to grayscale
4:      end if
5:      array ← convert img to 2D pixel array
6:      height ← number of rows in array
7:      width ← number of columns in array
8:      projection ← new array of size width
9:      for x = 0 to width − 1 do
10:         sum ← 0
11:         for y = 0 to height − 1 do
12:             sum ← sum + array[y][x]
13:         end for
14:         projection[x] ← sum
15:     end for
16:     return projection
17: end procedure
```

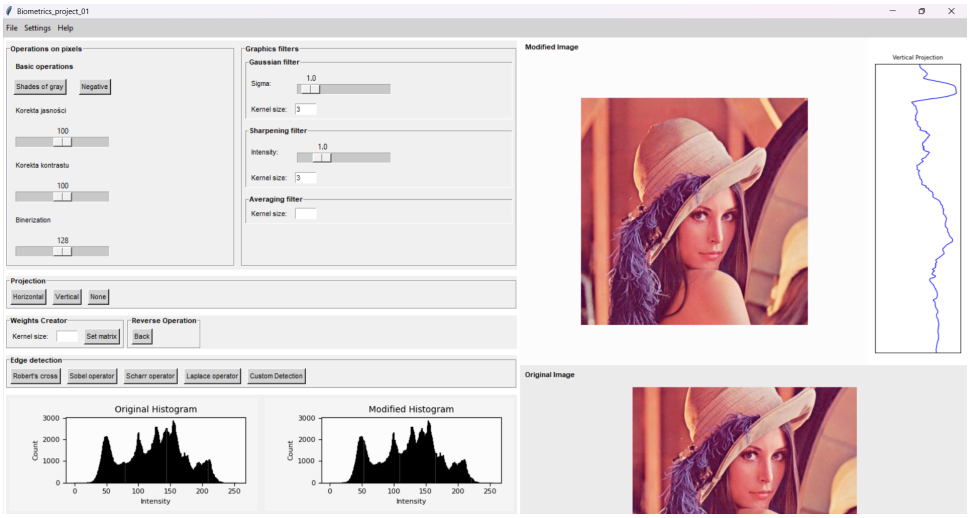

Figure 18: Above an example of application of the vertical projection for: lena.png.

Image Histograms:

In the left panel of the application, the user can see two intensity distribution histograms—one for the original image and one for the modified image. The histograms are computed by first converting the image to grayscale and calculating the luminance using the formula:

$$I = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

In this way, from three color channels (R, G, B), we obtain a single grayscale intensity value. After each image modification, the histogram is updated by counting, for each possible intensity level, how many pixels in the image have that specific value.

The algorithm itself is simple and intuitive, so it has not been included in the form of pseudocode.

Undo Functionality:

The final notable feature of the application is the ability to undo any modification. The solution is implemented in a technically simple manner. After each operation, a copy of the current image is pushed onto a stack. When an undo action is triggered, the last image from the stack is retrieved and restored.

4 Problems during implementation and boundaries

4.1 Problems encountered during application development

The primary problem was designing the GUI. There are libraries such as `PyQt` that significantly speed up and simplify the process of creating a graphical interface. However, instead of that, I used the default tkinter. It seems that it was not the best idea. Even when trying to load an image, there were problems with displaying it. The problem was

loading the image from the `PhotoImage` form from the PIL library and then updating the interface elements.

As for the loading itself, for some reason, loading the image only worked the second time. Meaning you had to load the image (at which point it did not appear), and then you had to load the same image again (only then did it appear on the screen). However, it did not respond to user modification attempts. The point is that after moving the sidebars or pressing some button to apply the change automatically, it often did not work. For example, binaryization with a given threshold worked, but applying the negative did not.

Another issue was that some of these operations worked with PNG files while others did not. Therefore, I decided that the application would support only JPG files, which is clearly indicated when the application is started. However, some operations also work with PNG, but to test the full capabilities of the application, JPG files should be loaded.

Another problem also concerned the GUI, specifically the projection and creating a layout for it. The assumptions were relatively simple. We have a modified image, and below it should be visible a horizontal projection plot, and next to it a vertical projection plot.

However, the main problem was that while it was possible to set fixed places for the vertical projection in the form of panels in the upper right corner of the application, that place would be empty if the user did not display the projection. For this reason, a dynamic layout was needed, which would change according to what the user wants to see on the image.

The problem was that in earlier versions the modified image was significantly stretched horizontally when displaying the horizontal projection and significantly shrunk when displaying the vertical projection. Ultimately, the project was partially resolved by moving the image to make room for the vertical projection plot. On the other hand, the original image shrinks.

It seems that this approach is original, since when looking at the projection plot, people look at the modified image rather than the original.

Another problem encountered was the desire to introduce some personalization to the application. The ultimate goal was for the user to be able to choose light/dark mode, set a custom color palette in the application, choose fonts, and have the option to display only the operations they wanted.

The only real issue here were aesthetic concerns. When trying to implement it, I concluded that the application was not actually more visually appealing afterward. Therefore, the `settings` option was ultimately removed, although some remnants remained in the code.

4.2 Application limitations

The application has two fundamentally significant limitations.

- The application is intended for JPG files.
- When undoing operations, we simply display the previous images, meaning that during each modification, we only save the image itself on the stack without the parameters associated with it, such as:
  - the binaryzation threshold at which the image was obtained,
  - the contrast applied,
  - the filter used,
  - the kernel size or settings etc.

  This has the limitation that when undoing operations, the sliders are not moved to the corresponding values. This may suggest to the user that the image is for the settings they see, although in reality, this is not true.

4.3 Summary

The project itself was quite interesting. There are definitely many aspects of the application that can still be improved.

First of all, a more user-friendly GUI could be developed. It would also be beneficial to add the ability to compare images at different stages of image modification, for instance, by adding a new window.

Another possible enhancement would be to add image upscaling to the application or generally integrate the application with an API of LLM providers to perform image operations automatically. The idea would be that the user writes a text describing the changes they want to apply to the image, and then the text is processed, and the operation is performed automatically.

This approach addresses the common situation where users know how the final image should look but are unsure about the intermediate steps or specific modifications needed to achieve that goal.