Project 2

Image processing

Student: Nguyen Minh Nhat
Student ID: 22127309
Class: 22CLC05
Applied Mathematics and Statistics
Department of Information and Techonology
University of Science, Viet Nam National University Ho Chi Minh City
July 2024

Abstract

This is a simple project to implement some basic image processing functions. The project is written in Python with the help of NumPy for matrix manipulation, Pillow for image processing, and Matplotlib for image visualization. The project is done as a part of the course ”Applied Mathematics and Statistics” at the University of Science, Viet Nam National University Ho Chi Minh City.

Contents

Acknowledgement **1**

1 Introduction **1**

2 Auxiliary Functions **2**

2.1 Read an image 2
2.2 Display an image 2
2.3 Save an image 2
2.4 Pad an image 2
2.5 Compare two images 2

3 Core Image Enhancement Algorithms **3**

3.1 Brightness Adjustment 3
3.2 Contrast Adjustment 3
3.3 Image Flipping 4
3.4 Color Transformation 4
3.5 RGB to Greyscale Conversion 5
3.6 RGB to Sepia Conversion 5
3.7 2D Convolution 5
3.8 Gaussian Blur (5x5) 6
3.9 Sharpen Filter 6
3.10 Center Cropping 7
3.11 Circular Cropping 8
3.12 Elliptical Cropping 8
3.13 Image Resizing Using Bilinear Interpolation 11
3.14 Main Function 12

4 Results **14**

References **18**

Acknowledgement

This project has been an incredible learning experience during my time at university. I have learned a lot from this project. I would like to express my gratitude toward my lecterers for this project. I am also grateful to StackOverflow, GitHub, and other websites for their invaluable resources. Special thanks to my classmates, Nguyen Long and Nguyen Trieu Khoang, for their brilliant ideas and insightful suggestions.

1 Introduction

This project focuses on implementing fundamental image processing functions using Python. The scope includes various operations such as adjusting brightness and contrast, image flipping, color space conversion, blurring and sharpening, as well as image cropping techniques. The implementation utilizes only essential libraries like PIL, NumPy, and Matplotlib, emphasizing the development of core image processing algorithms from scratch. This report outlines the methods, challenges, and results of implementing these image manipulation tasks.

2 Auxiliary Functions

2.1 Read an image

(function) read_img(img_path: str) → np.ndarray

This function reads an image from the specified path using PIL and converts it to a NumPy array.

2.2 Display an image

(function) show_img(img: np.ndarray) → None

Displays the given image using Matplotlib's imshow function.

2.3 Save an image

(function) save_img(img: np.ndarray, img_path: str) → None

Saves the given image to the specified path using PIL's Image module.

2.4 Pad an image

(function) pad_image(img: np.ndarray, target_shape: tuple) → np.ndarray

Pads the input image with zeros to reach the target shape, useful for image resizing.

2.5 Compare two images

(function) show_img_compare(img1: np.ndarray, img2: np.ndarray, title1: str, title2: str, axis: bool, space: int) → None

Displays two images side by side for comparison, with customizable titles and spacing.

3 Core Image Enhancement Algorithms

3.1 Brightness Adjustment

(function) adjust_brightness(img: np.ndarray, factor: float) → np.ndarray

Methodology: This function modifies the brightness of an image by adding a constant factor to all pixel values.

$$I_{\text{new}}(x, y) = I_{\text{original}}(x, y) + \text{factor}$$

The `np.clip` function is used to ensure the pixel values remain within the valid range of $[0, 255]$.

Weaknesses:

- The linear adjustment may lead to loss of detail in very bright or dark areas of the image.
- Extreme brightness adjustments can cause color distortion or loss of image quality.
- Adding the factor will increase the brightness of the image , but the image may appear with a slight veil as the contrast will be reduced. The gain(contrast) can be used to diminish this effect, but due to the saturation, some details in the original bright regions may be lost. The effect also occurs when the factor is negative. Gamma correction can be used to correct the brightness of an image [1].

3.2 Contrast Adjustment

(function) adjust_contrast(img: np.ndarray, factor: float) → np.ndarray

Methodology: This function modifies the contrast of an image by multiplying all pixel values by a constant factor. The operation is applied element-wise across the entire image array.

$$I_{\text{new}}(x, y) = I_{\text{original}}(x, y) \times \text{factor}$$

The `np.clip` function is used to ensure the pixel values remain within the valid range of $[0, 255]$.
Weaknesses:

- The multiplicative adjustment may exaggerate noise in dark areas of the image.
- Extreme contrast adjustments can lead to loss of detail in highlights and shadows.

3.3 Image Flipping

(function) flip_image(img: np.ndarray, is_horizontal: bool) → np.ndarray

Methodology: This function reverses the order of pixels along either the horizontal or vertical axis of the image. The operation is performed well as the use of NumPy slicing, which provides a *view* of the original array without copying the data without any loss of information. [2]

$$I_{\text{new}}(x, y) = \begin{cases} I_{\text{original}}[:, :: -1] & \text{if } \textit{is\_horizontal} \\ I_{\text{original}}[:: -1, :] & \text{otherwise} \end{cases}$$

Weaknesses:

- Limited to only horizontal or vertical flipping, not arbitrary rotations.
- Although fast, changes in the original can lead to changes in the output as NumPy slicing creates a view instead of a copy. Care must be taken to explicitly copy the array when necessary to avoid unintended modifications to the original data [2].

3.4 Color Transformation

(function) apply_color_transform(image: np.ndarray, weights: np.ndarray) → np.ndarray

Methodology: This function applies a color transformation to an image using the provided weights for the RGB channels. The operation is similar to `np.dot`, but more readable to the writer and faster in pure NumPy without optimized libraries like BLAS or MKL. The transformation is performed using NumPy's `einsum` function, which efficiently handles the matrix multiplication.

$$I_{\text{new}} = \text{einsum}('ijk, kl- > ijl', I_{\text{original}}, W^T)$$

Which can be understood as multiplying the RGB channels of the I_{original} image `(i, j, k)` with the transpose of weights matrix `(k, l)` then sum over the `('k')` axis to get the I_{new} with the shape of `(i, j, l)`.
Additionally, the matrix multiplication can be expressed as:

$$I_{\text{new}} = I_{\text{original}} \cdot W^T$$

The I_{new} is then clipped to ensure the transformed values are within the valid range for image data.
Weaknesses: `np.dot` may be faster for large matrices, especially when using optimized libraries like BLAS or MKL.

3.5 RGB to Greyscale Conversion

(function) rgb2grey(img: np.ndarray) → np.ndarray

Methodology: This function converts an RGB image to a greyscale image using a specific set of weights for the RGB channels.

$$W_{\text{grey}} = \begin{bmatrix} 0.2125 & 0.7154 & 0.0721 \\ 0.2125 & 0.7154 & 0.0721 \\ 0.2125 & 0.7154 & 0.0721 \end{bmatrix} \qquad [3]$$

The transformation is applied using the `apply_color_transform` function.

3.6 RGB to Sepia Conversion

(function) rgb2sepia(img: np.ndarray) → np.ndarray

Methodology: This function converts an RGB image to a sepia-toned image using a specific set of weights for the RGB channels.

$$W_{\text{sepia}} = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix} \qquad [4]$$

The transformation is applied using the `apply_color_transform` function.

3.7 2D Convolution

(function) convolution_2d(image: np.ndarray, kernel: np.ndarray) → np.ndarray

Methodology: This function performs a 2D convolution on an image using a given kernel. The convolution operation is applied separately to each color channel of the image to prevent excessive memory allocation while maintaining performance.

First, the image is padded to handle the borders by using `np.pad` with the `edge` mode, which minimizes the effect of the padding on the convolution result comaprared to other modes like `constant` or `reflect`. . The kernel is reshaped into a vector, and the image is divided into patches corresponding to the kernel size.
For each channel, windows with kernel shape are extracted from the padded image using `np.lib.stride_tricks.as_strided`. These windows are then flattened and multiplied element-wise with the kernel vector in the next step:

$$I_{\text{result}}[:, :, c] = \text{einsum}('ijk, k- > ij', \text{windows}, \text{kernel})$$

This operation can be understood as extracting patches from each channel of the padded image (indexed by i, j, k) and performing an element-wise multiplication with the kernel (indexed by k), then summing over the k axis to produce the convolved image for each channel (indexed by i, j).

Finally, the resulting image is clipped to ensure the pixel values are within the valid range (0 to 255) and converted to an unsigned 8-bit integer type to match the standard image format.

$$I_{\text{final}} = \text{np.clip}(I_{\text{result}}, 0, 255).astype(\text{np.uint8})$$

Main operation can be done more efficiently by applying kernel to three color channels at the same time, but this approach may require more memory and processing power for large images or kernels. Separating the channels allows for less memory allocation and the readability exchange for a slight decrease in performance.
Weaknesses: For very large images or kernels, the memory usage and performance may still be a concern, but the channel-wise processing helps mitigate this issue by not requiring all color channels to be processed simultaneously.

3.8 Gaussian Blur (5x5)

(function) apply_gaussian_blur_5x5(image: np.ndarray) → np.ndarray

Methodology: This function applies a 5x5 Gaussian blur to an image. The kernel used for the Gaussian blur is defined as follows:

$$K_{\text{gaussian}} = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \qquad [5]$$

The convolution operation is performed using the `convolution_2d` function.
Weaknesses: The effect only visible when the image has low resolution. Improvement can be made by using a larger kernel size or applying multiple passes of the filter, but this will increase the computational cost and may lead to oversmoothing.

3.9 Sharpen Filter

(function) apply_sharpen_filter(image: np.ndarray) → np.ndarray

Methodology: This function applies a sharpen filter to an image. The kernel used for the sharpen filter is defined as follows:

$$K_{\text{sharpen}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad [5]$$

The convolution operation is performed using the `convolution_2d` function.
Weaknesses: The sharpen filter can hardly be seen in the image with high resolution. The effect can be improved by using a larger kernel size or applying multiple passes of the filter, but this will increase the computational cost and may lead to over-sharpening.

3.10 Center Cropping

(function) crop_center(image: np.ndarray, percentage: float) → np.ndarray

Methodology: This function crops an image from the center. The percentage parameter defines how much of the image should be kept, with valid values in the range (0, 1] (cases where the percentage is zero would result in an empty image).
If the percentage value is outside this range, it is clipped to the nearest valid value, and a message is printed indicating the correction.
The cropping factor is calculated as:

$$\text{factor} = \frac{1}{\sqrt{\text{percentage}}} = \left(\frac{1}{\text{percentage}}\right)^{0.5}$$

The new dimensions are then determined by:

$$h_{\text{new}} = \frac{h}{\text{factor}}, \quad w_{\text{new}} = \frac{w}{\text{factor}}$$

where h and w are the original height and width of the image. The cropping operation is centered based on the calculation of the starting and ending indices for the new image, then applied to the original image with `NumPy slicing`:

$$\begin{aligned} h_{\text{start}} &= \frac{h - h_{\text{new}}}{2}, \quad h_{\text{end}} = h_{\text{start}} + h_{\text{new}} \\ w_{\text{start}} &= \frac{w - w_{\text{new}}}{2}, \quad w_{\text{end}} = w_{\text{start}} + w_{\text{new}} \\ I_{\text{new}} &= I_{\text{original}}[h_{\text{start}} : h_{\text{end}}, w_{\text{start}} : w_{\text{end}}] \end{aligned}$$

Weaknesses:

- The cropping area is always a square
- Slicing is fast and efficient, but it provides a view of the original array, which may lead to unintended modifications if not handled carefully. [2]

3.11 Circular Cropping

(function) circular_crop(image: np.ndarray) → np.ndarray

Methodology: This function crops a 2D image to a circular region centered in the image. The circle is centered in the image and has a radius equal to half the smaller dimension of the image.
The equation used to check if a point (x, y) lies inside the circle is:

$$(x - x_c)^2 + (y - y_c)^2 \leq r^2$$

where (x_c, y_c) is the center of the circle, and r is the radius.
In the implementation, the center of the image is calculated as:

$$\text{center} = \frac{\begin{bmatrix} \text{height} + 1 \\ \text{width} + 1 \end{bmatrix}}{2}$$

The square of the radius is:

$$\text{square\_radius} = (\min(\lfloor \text{center} \rfloor))^2$$

A grid of indices representing the coordinates of each pixel in the image is created, and the distance of each pixel from the center is calculated as:

$$\text{diff} = \text{grid} - \text{center}$$

The distance is then squared and summed along the last axis to obtain the squared distance from the center using einsum:

$$(x - x_c)^2 + (y - y_c)^2 = \text{np.einsum}('ijk, ijk- > jk', \text{diff}, \text{diff})$$

Finally, the mask of pixels inside the circle, which is created by checking if the squared distance is less than or equal to the square of the radius, is then applied to the original image by element-wise multiplication.
Weaknesses: The circular cropping may lead to aliasing artifacts for low resolution images or images with sharp edges.

3.12 Elliptical Cropping

(function) ellipses_crop(image: np.ndarray) → np.ndarray

Methodology: This function crops a 2D image to an elliptical region centered in the image. The ellipse is centered in the image and has axes that are 60% of the height and 36.5% of the width of the image.
The ellipse equation in the general form is:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

where (x_c, y_c) is the center of the ellipse, a and b are the semi-major and semi-minor axes, respectively. To rotate the ellipse by an angle θ, rotation matrix is used:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{[6]}$$

Let (x', y') be the coordinates of a point in the rotated ellipse, and (x, y) be the coordinates of the same point in the original ellipse. The rotation matrix can be applied as:

$$\begin{aligned} \begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &= \begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ x\sin(\theta) + y\cos(\theta) \end{bmatrix} \end{aligned}$$

The new ellipse equation after rotation to check if a point (x, y) lies inside the ellipse is:

$$\left(\frac{(x - x_c)\cos(\theta) + (y - y_c)\sin(\theta)}{a}\right)^2 + \left(\frac{(x - x_c)\sin(\theta) - (y - y_c)\cos(\theta)}{b}\right)^2 \leq 1$$

where (x_c, y_c) is the center of the ellipse, a and b are the semi-major and semi-minor axes, respectively, and θ is the rotation angle.

In the implementation, the center of the image is calculated as:

$$\text{center} = \frac{\begin{bmatrix} \text{height} + 1 \\ \text{width} + 1 \end{bmatrix}}{2}$$

The axes of the ellipse are:

$$\text{axes} = \begin{bmatrix} \text{height} \times 0.6 \\ \text{width} \times 0.365 \end{bmatrix}$$

A grid of indices representing the coordinates of each pixel in the image is created, and the distance of each pixel from the center is calculated as:

$$\text{diff} = \text{grid} - \text{center}$$

Using a rotation factor $rf = \frac{1}{\sqrt{2}}$ corresponding to a 45-degree rotation.

First ellipse equation:

$$\left(\frac{(x - x_c) \times rf + (y - y_c) \times rf}{a}\right)^2 + \left(\frac{(x - x_c) \times rf - (y - y_c) \times rf}{b}\right)^2 \leq 1$$

The squared distance from the center of the ellipse is calculated for the first ellipse rotation:

$$\begin{aligned}\text{first\_a\_rot} &= \left(\frac{(x-x_c)\times rf+(y-y_c)\times rf}{a}\right)^2\\ &= \text{np.einsum}('ijk, i->jk', \text{diff}, \frac{[rf, rf]}{axes[0]})^2\\ \text{first\_b\_rot} &= \left(\frac{(x-x_c)\times rf-(y-y_c)\times rf}{b}\right)^2\\ &= \text{np.einsum}('ijk, i->jk', \text{diff}, \frac{[-rf, rf]}{axes[1]})^2\end{aligned}$$

Second ellipse equation:

$$\left(\frac{(x-x_c)\times rf-(y-y_c)\times rf}{a}\right)^2+\left(\frac{-(x-x_c)\times rf-(y-y_c)\times rf}{b}\right)^2\leq 1$$

The squared distance from the center of the ellipse is calculated for the second ellipse rotation:

$$\begin{aligned}\text{second\_a\_rot} &= \left(\frac{(x-x_c)\times rf-(y-y_c)\times rf}{a}\right)^2\\ &= \text{np.einsum}('ijk, i->jk', \text{diff}, \frac{[-rf, rf]}{axes[0]})^2\\ \text{second\_b\_rot} &= \left(\frac{-(x-x_c)\times rf-(y-y_c)\times rf}{b}\right)^2\\ &= \text{np.einsum}('ijk, i->jk', \text{diff}, \frac{[-rf, -rf]}{axes[1]})^2\end{aligned}$$

Finally, the mask of pixels inside the ellipse, which is created by checking if the squared distance is less than or equal to 1, is then applied to the original image by element-wise multiplication:

$$\text{mask} = ((\text{first\_a\_rot}+\text{first\_b\_rot})\leq 1)\,|\,((\text{second\_a\_rot}+\text{second\_b\_rot})\leq 1)$$

Weaknesses:

- The elliptical cropping may introduce artifacts at the boundary of the ellipse, especially in low-resolution images.
- $(0.6, 0.365)$ are the ratios that manually set to create the ellipse without any mathematical proof, may not be perfectly aligned
- Implementation only intended to work with square images.

3.13 Image Resizing Using Bilinear Interpolation

(function) resize(image: np.ndarray, new_height: int, new_width: int)
→ np.ndarray

Methodology: This function resizes a 3D image to a new height and width using bilinear interpolation.
Bilinear interpolation calculates the new pixel values by considering the four neighboring pixels from the original image and their distances to the target pixel.
Let (x', y') be the coordinates of a pixel in the new image, and (x, y) be the corresponding coordinates in the original image. The four neighboring pixels in the original image are:

- (x_1, y_1): the top-left neighbor
- (x_2, y_1): the top-right neighbor
- (x_1, y_2): the bottom-left neighbor
- (x_2, y_2): the bottom-right neighbor

The interpolated pixel value $I(x', y')$ calculated as:

$$I(x', y') \approx w_{11}I(x_1, y_1) + w_{21}I(x_2, y_1) + w_{12}I(x_1, y_2) + w_{22}I(x_2, y_2) \qquad [7]$$

where:

- $I(x_1, y_1)$: intensity of the top-left neighbor
- $I(x_2, y_1)$: intensity of the top-right neighbor
- $I(x_1, y_2)$: intensity of the bottom-left neighbor
- $I(x_2, y_2)$: intensity of the bottom-right neighbor
- $w_{11} = (x_2 - x')(y_2 - y')/(x_2 - x_1)(y_2 - y_1)$
- $w_{21} = (x' - x_1)(y_2 - y')/(x_2 - x_1)(y_2 - y_1)$
- $w_{12} = (x_2 - x')(y' - y_1)/(x_2 - x_1)(y_2 - y_1)$
- $w_{22} = (x' - x_1)(y' - y_1)/(x_2 - x_1)(y_2 - y_1)$

In this implementation:
The scale factor is calculated as:

$$\text{scale} = \begin{bmatrix} \text{old\_height}/\text{new\_height} \\ \text{old\_width}/\text{new\_width} \end{bmatrix}$$

Using the scale factor, the indices of the new image are calculated as:

$$y', x' = \text{np.indices}((\text{new\_height}, \text{new\_width})) \times \text{scale}$$

The indices are then flattened, use the `np.floor` function to get the top-left pixel indices, and the `np.minimum` function to get the bottom-right pixel indices:

$$\begin{aligned}
x_1 &= \text{np.floor}(x).astype(\text{np.uint32}) \\
y_1 &= \text{np.floor}(y).astype(\text{np.uint32}) \\
x_2 &= \text{np.minimum}(x_1 + 1, \text{old\_width} - 1) \\
y_2 &= \text{np.minimum}(y_1 + 1, \text{old\_height} - 1)
\end{aligned}$$

For each color channel `i`, the four neighboring pixels are extracted from the original image. The weights for interpolation are calculated, ignoring the division part. We assume $(x_2 - x_1)(y_2 - y_1) = 1$ to simplify the process and avoid division by zero, which is valid most of the time:

$$\begin{aligned}
I_a &= \text{image}[y_1, x_1, i] \\
I_b &= \text{image}[y_2, x_1, i] \\
I_c &= \text{image}[y_1, x_2, i] \\
I_d &= \text{image}[y_2, x_2, i] \\
w_a &= (x_2 - x) \times (y_2 - y) \\
w_b &= (x_2 - x) \times (y - y_1) \\
w_c &= (x - x_1) \times (y_2 - y) \\
w_d &= (x - x_1) \times (y - y_1)
\end{aligned}$$

The color channel of the new image is calculated as:

$$\text{new\_image}[:, :, i] = w_a I_a + w_b I_b + w_c I_c + w_d I_d$$

Calculating through each color channel can reduce the memory usage and increase the readability of the code, but it may slightly decrease the performance compared to applying the operation to all channels at once.

3.14 Main Function

(function) main() → None

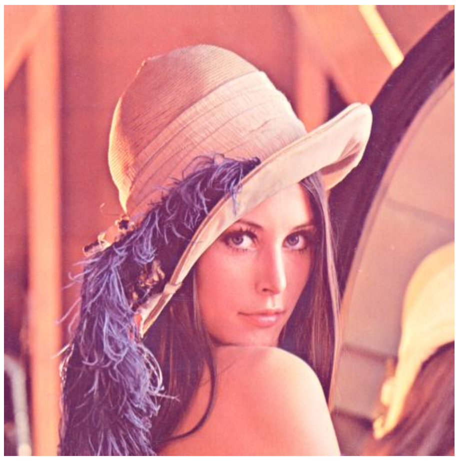
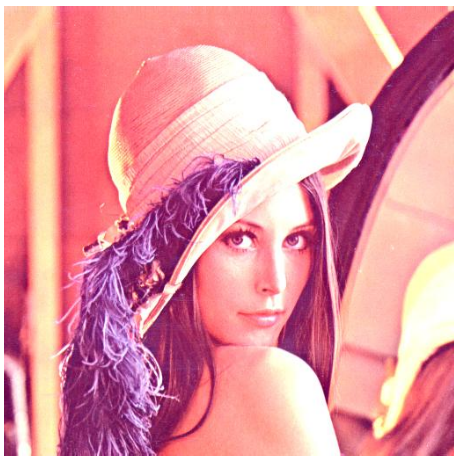
The main function allows the user to interactively apply various image processing operations to an image. The user is prompted to enter the path of the image to process and the extension of the output image. The user can then select an operation to perform from a list of options. The function reads the image, applies the selected operation, displays the original and processed images, and saves the processed image with the specified extension. The user can continue to select and apply operations until they choose to exit the program.

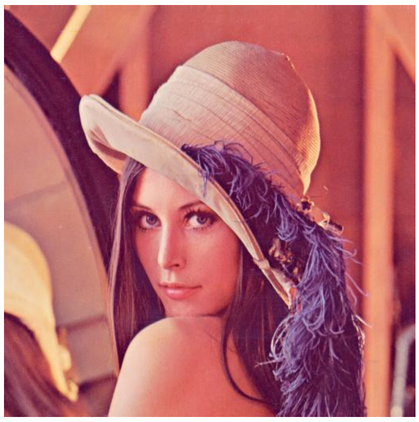
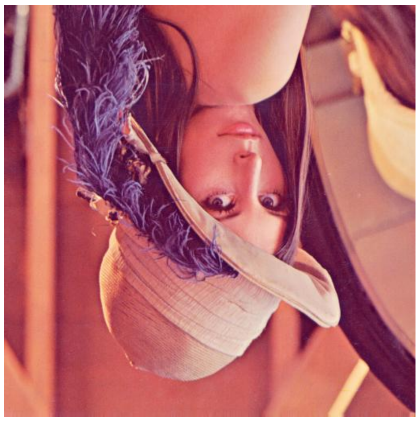
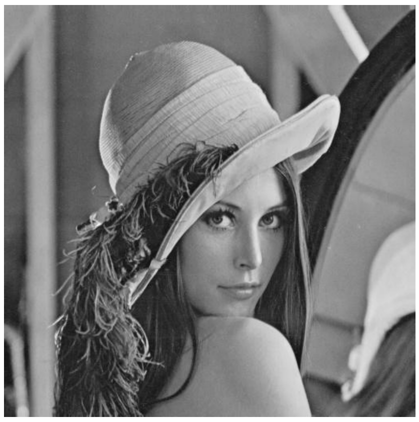
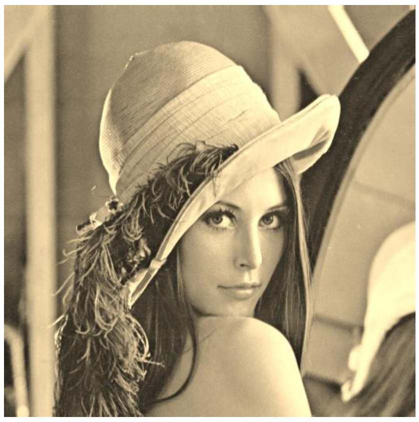
There are 14 operations available:

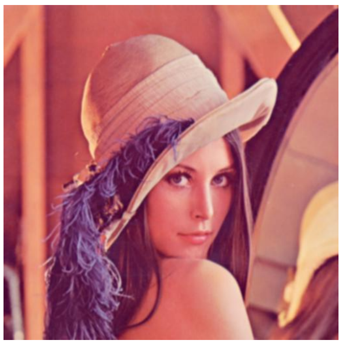
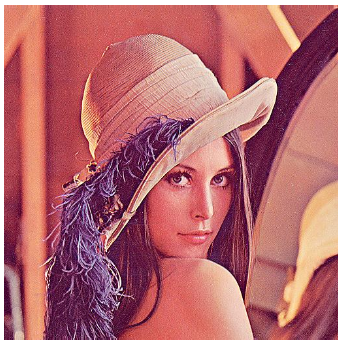
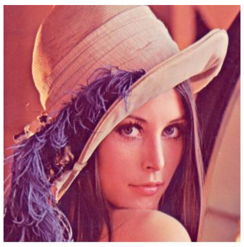
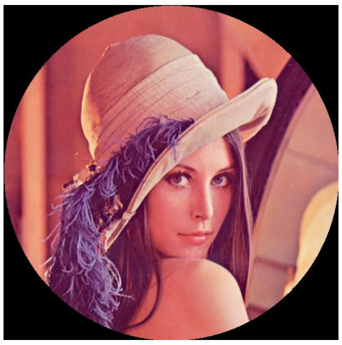
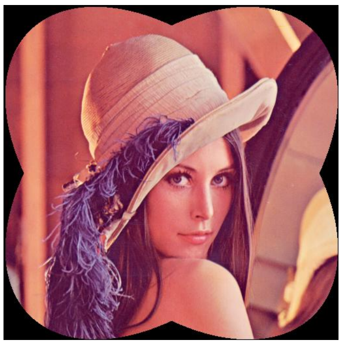
- 0. Do everything, number and factor will be assigned by the program
- 1. Change Brightness
- 2. Change Contrast
- 3. Flip Horizontally
- 4. Flip Vertically
- 5. RGB to Grayscale
- 6. RGB to Sepia
- 7. Blur Image
- 8. Sharpen Image
- 9. Crop Image by Size
- 10. Circular Crop
- 11. Elliptical Crop
- 12. Zoom In/Out
- 13. Reset

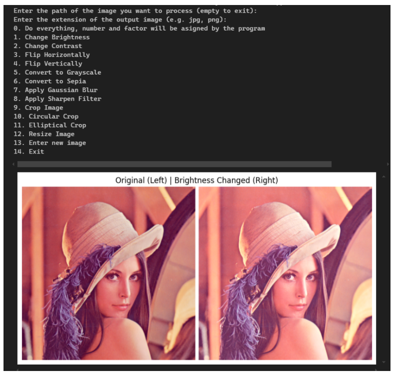
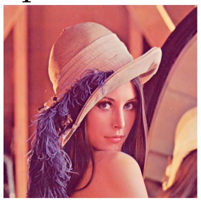
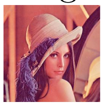
Weaknesses: Jupyter Notebook is not suitable for interactive user input, which may lead to errors when running the main function.

4 Results

No.	Operation	Completion Level	Output
	Original Image		[8]
1	Change Brightness	100%	
2	Change Contrast	100%	

No.	**Operation**	**Completion Level**	**Output**
3.1	Flip Horizontally	100%	
3.2	Flip Vertically	100%	
4.1	RGB to Grayscale	100%	
4.2	RGB to Sepia	100%	

No.	Operation	Completion Level	Output
5.1	Blur Image	100%	
5.2	Sharpen Image	100%	
7.1	Crop Image by Size	100%	
7.2	Circular Crop	100%	
7.3	Elliptical Crop	100%	

No.	**Operation**	**Completion Level**	**Output**
8	main function	100%	 Enter the path of the image you want to process (empty to exit): Enter the extension of the output image (e.g. jpg, png): 0. Do everything, number and factor will be asigned by the program 1. Change Brightness 2. Change Contrast 3. Flip Horizontally 4. Flip Vertically 5. Convert to Grayscale 6. Convert to Sepia 7. Apply Gaussian Blur 8. Apply Sharpen Filter 9. Crop Image 10. Circular Crop 11. Elliptical Crop 12. Resize Image 13. Enter new image 14. Exit
9	Zoom In/Out 2x	100%	upscaled to 1024 x 1024  original size 512 x 512  downscaled to 256 x 256

References

[1] OpenCV Documentation. Basic linear transformations. `https://docs.opencv.org/4.x/d3/dc1/tutorial_basic_linear_transform.html`. [Accessed: 29th July 2024].

[2] NumPy. Indexing - numpy manual. `https://numpy.org/doc/stable/user/basics.indexing.html#basics-indexing`. [Accessed: 29th July 2024].

[3] Scikit-image Contributors. Rgb to gray conversion. `https://scikit-image.org/docs/stable/auto_examples/color_exposure/plot_rgb_to_gray.html`. [Accessed: 29th July 2024].

[4] Luis Pedro Coelho. Mahotas library - rgb to sepia conversion. `https://github.com/luispedro/mahotas/blob/00b8c9748e3fc01d794847c67b30d49a7175833c/mahotas/colors.py#L186`. [Accessed: 30th July 2024].

[5] Wikipedia. Kernel (image processing). `https://en.wikipedia.org/wiki/Kernel_(image_processing)`, . [Accessed: 30th July 2024].

[6] University of Illinois at Urbana-Champaign. Coordinate transformations. `https://courses.illinois.edu/matse484/CoordinateTransformations.html`. [Accessed 30 July 2024].

[7] Wikipedia. Bilinear interpolation. `https://en.wikipedia.org/wiki/Bilinear_interpolation`, . [Accessed: 30th July 2024].

[8] Github Contributors: mikolalysenko. The lenna image. `https://github.com/mikolalysenko/lena/blob/master/lena.png`. [Accessed: 30th July 2024].