UNIVERSITY OF SCIENCE – VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

FACULTY OF INFORMATION TECHNOLOGY – HIGH-QUALITY PROGRAM

PROJECT REPORT - APPLIED MATHEMATICS AND STATISTICS FOR INFORMATION TECHNOLOGY

IMAGE PROCESSING

Class: 23CLC05

Theory Instructor: Vu Quoc Hoang

Practical Instructors: Tran Thi Thao Nhi, Nguyen Ngoc Toan

Student Name: Le Hong Ngoc

Student ID: 23127236

Ho Chi Minh City, July, 2025

Contents

I. Project Information

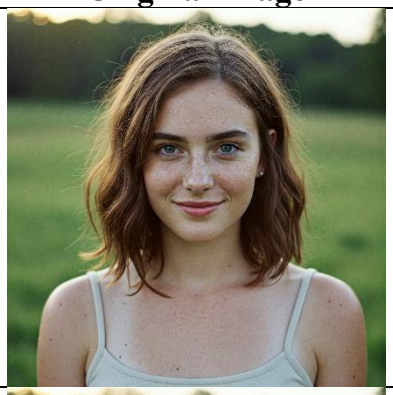
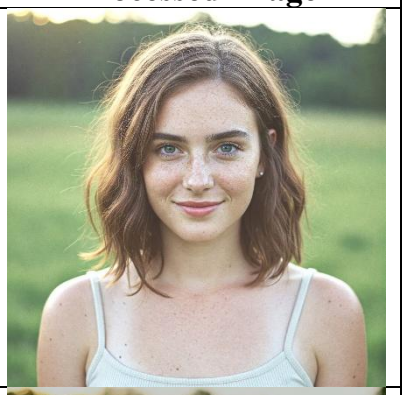
In the previous project, we were introduced to the concept that an image is stored as a matrix of pixels. Each pixel can either be a single value (grayscale image) or a vector (color image). In this project, we will apply the matrix operations we have learned to implement the following basic image processing functions:

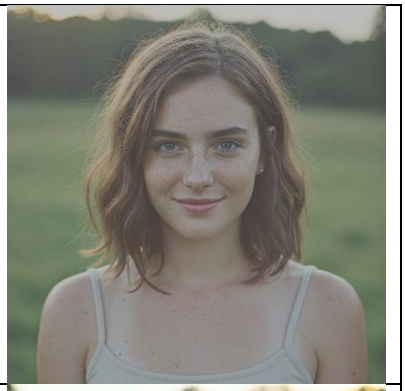
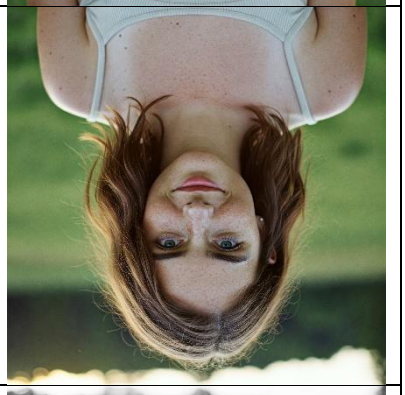
- Adjust (increase/decrease) the brightness of the image
- Adjust (increase/decrease) the contrast of the image
- Flip the image horizontally/vertically
- Convert an RGB image to grayscale/sepia
- Blur/sharpen the image
- Crop 1/4 of the image by size (cropped from the center)
- Crop the image using a mask (circle/two intersecting ellipses)

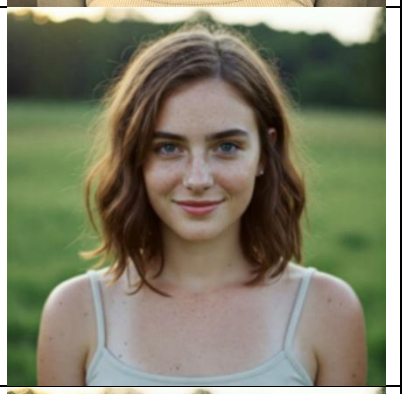
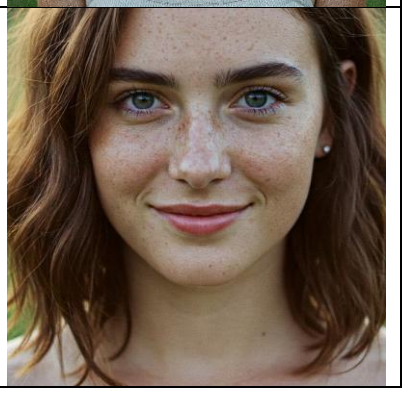
II. Completed Functions

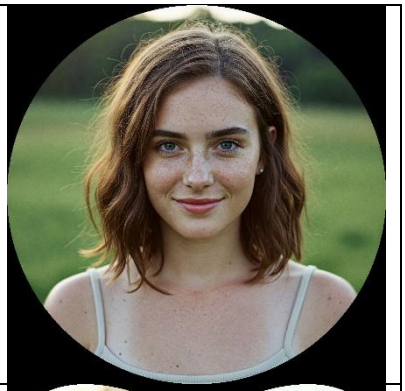
Below are the results after performing the image processing functions. I have selected two images: the first is a square image with dimensions 1280x1280, and the second is a rectangular image with dimensions 1280x853.

- Square image - 1280x1280:

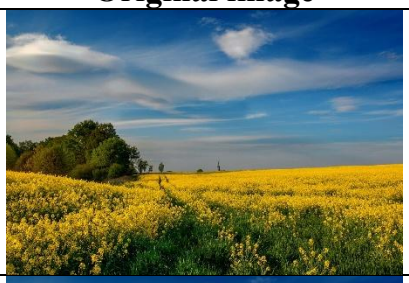
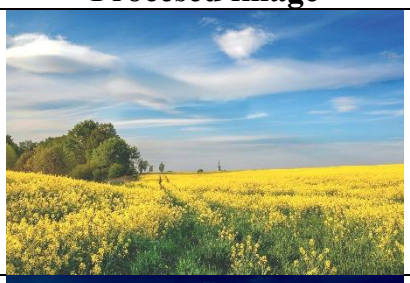
No.	Function	Original image	Processed image
1.1	Increase image brightness (+50)		
1.2	Decrease image brightness (-50)		
2.1	Increase image contrast (+1.5)		

2.2	Decrease image contrast (+0.5)		
3.1	Flip image horizontally		
3.2	Flip image vertically		
4.1	Convert RGB image to grayscale		

4.2	Convert RGB image to sepia		
5.1	Blur image		
5.2	Sharpen image		
6	Crop 1/4 of the image (from center)		

7.1	Crop image with circular mask		
7.2	Crop image with two intersecting ellipses mask		

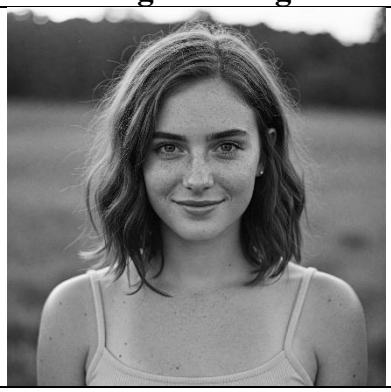
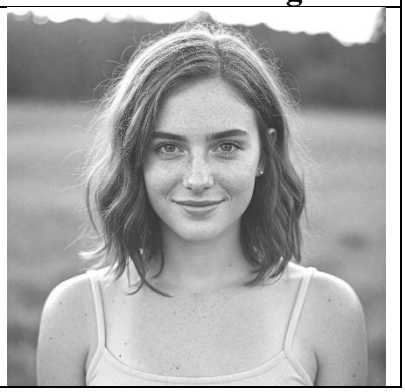
- Rectangular image – 1280x853:

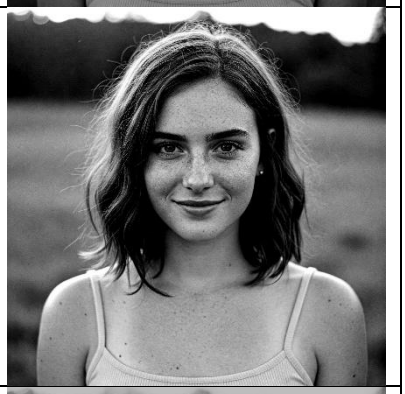
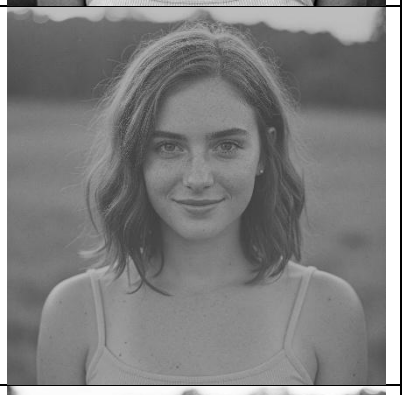
No.	Function	Original image	Procesed image
1.1	Increase image brightness (+50)		
1.2	Decrease image brightness (-50)		
2.1	Increase image contrast		

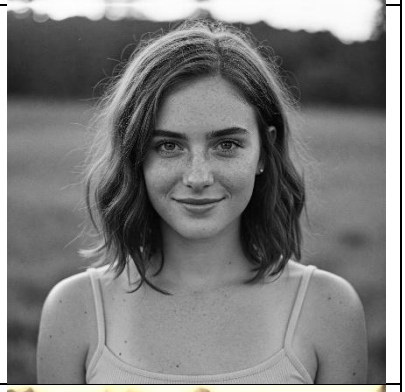
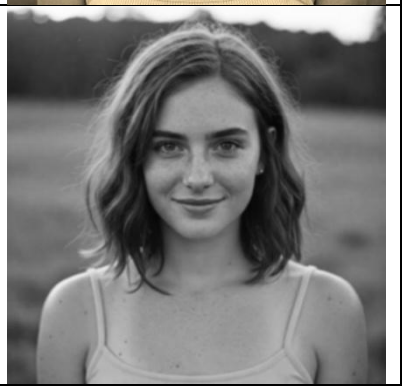
2.2	Decrease image contrast (+0.5)		
3.1	Flip image horizontally		
3.2	Flip image vertically		
4.1	Convert RGB image to grayscale		
4.2	Convert RGB image to sepia		
5.1	Blur image		

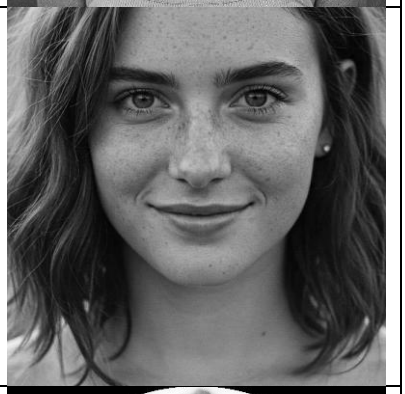
5.2	Sharpen image		
6	Crop 1/4 of the image (from center)		
7.1	Crop image with circular mask		
7.2	Crop image with two intersecting ellipses mask		

In addition, the following are the results of the functions when applied to a grayscale input image.

No.	Function	Original image	Processed image
1.1	Increase image brightness (+50)		

1.2	Decrease image brightness (-50)		
2.1	Increase image contrast (+1.5)		
2.2	Decrease image contrast (+0.5)		
3.1	Flip image horizontally		

3.2	Flip image vertically		
4.1	Convert RGB image to grayscale		
4.2	Convert RGB image to sepia		
5.1	Blur image		

5.2	Sharpen image		
6	Crop 1/4 of the image (from center)		
7.1	Crop image with circular mask		
7.2	Crop image with two intersecting ellipses mask		

III. The implemented functions

In this section, I will detail the 11 functions implementing the 7 image-processing features described earlier, along with 1 *process_image()* function, which acts as the central orchestrator for the entire processing pipeline. In addition, the program includes helper functions for reading, saving, and displaying images; however, these utility functions were already covered in the previous project, so I will not repeat them here. The program also includes a main function, which serves as the primary user interface: it takes user input and invokes process_image() to apply the selected operations.

Below are the details of the 12 functions mentioned above:

1. adjust_brightness()

This function performs the operation of adjusting (increasing or decreasing) the brightness of an image, with the input parameter *img_2d* – a 2D NumPy array representing the image.

1.1. Impletementation Idea

As we know, each RGB color image is represented by a matrix of pixels, where each pixel stores three values corresponding to the R, G, and B color channels, with values ranging from 0 to 255.

Therefore, to adjust the brightness of an RGB image, we simply increase (or decrease) the pixel values that represent the image. Adding a larger positive number makes the image brighter, while adding a negative number makes the image darker [10].

1.2. Impletementation Details

- The brightness level can be adjusted using the *brightness* variable (*brightness* > *0* increases brightness, *brightness* < 0 decreases it, and *brightness* = 0 leaves it unchanged).
- Before adding a value to each pixel, the array *img_2d* is converted to type *float32* using *astype()* to avoid overflow errors.
- The *clip()* function from NumPy is used to ensure that all pixel values remain within the valid range of [0, 255].
- Finally, the array is cast back to *uint8* so the image displays with correct colors when saved or shown.

2. adjust_contrast()

This function performs the operation of adjusting (increasing or decreasing) the contrast of an image, with the input parameter *img_2d* – a 2D NumPy array representing the image.

2.1. Impletementation Idea

Image contrast reflects the degree of difference between bright and dark regions in a picture. It is essentially the magnitude of the gap between the brightest and darkest pixels. Therefore, to adjust the contrast (either increase or decrease), we modify the difference between light and dark pixels. Specifically, we change the pixel values by scaling their distance from the midpoint of the value range, which is 127.5 (since pixel intensity ranges from 0 to 255). This adjustment is performed using the formula [10]:

$$f(x) = \alpha(x - 127.5) + 127.5$$

This formula adjusts the contrast by shifting pixel values toward the average level of 127.5, then increasing or decreasing them by multiplying with the factor α. If $\alpha > 1$, it increases contrast, while $0 < \alpha < 1$ reduces it. Finally, 127.5 is added back to bring the pixel values into the original range.

2.2. Impletementation Details

- The contrast level can be adjusted using the *contrast* variable (*contrast* > 1 increases contrast, 0 < *contrast* < 1 decreases it).
- Before applying the contrast adjustment formula, the array *img_2d* is converted to type *float32* using *astype()* to avoid overflow errors.
- The *clip()* function from NumPy is used to ensure that all pixel values remain within the valid range of [0, 255].
- Finally, the array is cast back to *uint8* so the image displays with correct colors when saved or shown.

3. flip_image_horizontal() and flip_image_vertical()

These two functions perform the operation of flipping the image horizontally and vertically, with the input parameter *img_2d* – a 2D NumPy array representing the image.

3.1. Impletementation Idea

Image flipping is the process of reversing the order of pixels along a specific axis.

When flipping horizontally (left ↔ right): we reverse the order of columns in each row, which corresponds to reversing pixels along the horizontal axis (axis=1).

When flipping vertically (top ↕ bottom): we reverse the order of rows, which corresponds to reversing pixels along the vertical axis (axis=0).

3.2. Impletementation Details

Use the *flip()* function from the NumPy library to reverse the array along the desired axis:

- *axis* = 1: to flip the array horizontally
- *axis* = 0: to flip it vertically.

4. convert_rgb_to_gray() and convert_rgb_to_sepia()

These two functions perform the operation of converting the RGB image to the grayscale and sepia image, with the input parameter *img_2d* – a 2D NumPy array representing the image.

4.1. Impletementation Idea

a. Conversion from RGB to Grayscale: convert_rgb_to_gray()

When converting an RGB color image to grayscale, we need to combine the three color channels R, G, B into a single value that represents the brightness of each pixel. According to studies, the human eye is most sensitive to green light, followed by red, and least sensitive to blue. Therefore, different weights are applied to each color channel to better reflect human visual perception.

A widely used formula based on the standard Rec.709 color space used for digital displays [8]:

$$Gray = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$$

This formula is applied to every pixel in the image to obtain the corresponding grayscale image matrix.

b. Conversion from RGB to Sepia: convert_rgb_to_sepia()

Similar to the process of converting an RGB image to grayscale, converting it to sepia also involves transforming the values of the three RGB color channels. However, unlike grayscale images, sepia-toned images retain all three color channels, but with altered intensities to create a warm, antique-like tone.

Each pixel in the RGB image is transformed using the following formula [7]:

$$TR = 0.393 \cdot R + 0.769 \cdot G + 0.189 \cdot B$$
$$TG = 0.349 \cdot R + 0.686 \cdot G + 0.168 \cdot B$$
$$TB = 0.272 \cdot R + 0.534 \cdot G + 0.131 \cdot B$$

Here, TR, TG, and TB represent the transformed values for the R, G, B channels of the sepia image. After computing these new values for each pixel, the resulting image will have a brownish tone characteristic of sepia image.

4.2. Impletementation Details

a. Conversion from RGB to Grayscale: convert_rgb_to_gray()

- The *dot()* function from the NumPy is used to perform a dot product between each RGB pixel in the *img_2d* matrix and a weight vector, according to the grayscale conversion formula provided above.

- After the dot product, the result is a grayscale image matrix of shape (H, W). To facilitate image display and saving in RGB format (H, W, 3), the *stack()* function from NumPy is used to replicate the grayscale channel into three identical channels, forming an RGB image with equal gray values.
- Finally, *astype()* is used to cast the data type to *uint8*, ensuring that the image is displayed with correct colors when saved or shown.

b. Conversion from RGB to Sepia: convert_rgb_to_sepia()

- A 3x3 matrix *sepia_img* is created, containing the coefficients used to convert an RGB image to a sepia-toned image based on the formula provided above.
- NumPy performs broadcasting to multiply each RGB pixel of the image matrix *img_2d* with the transpose of the *sepia_img* matrix. That is, for each pixel [R, G, B] the following mathematical operation is applied:

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} R & G & B \end{bmatrix} \cdot \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix}$$

- Since the matrix multiplication might produce values outside the valid pixel range [0, 255], *clip()* function is used to constrain the results within that range. Afterward, *astype()* is used to cast the data type to *unit8*, ensuring the image is displayed correctly when saved or shown.

5. blur_image() and sharpen_image()

These two functions perform the operation of blurring and sharpening an image, with the input parameter *img_2d* – a 2D NumPy array representing the image.

5.1. Impletementation Idea

a. Image blurring: blur_image()

- In image processing, a kernel (also known as a convolution matrix or filter mask) is a small matrix used to perform various image operations such as blurring, sharpening, embossing, edge detection,... This is achieved by performing a convolution between the kernel and the input image. Simply put, each pixel in the output image is calculated as a function of its neighboring pixels (including itself) in the input image, and the kernel serves as that function. The general expression of a convolution is [6]:

$$g(x,y) = \omega * f_{x,y} = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \omega_{i,j} \cdot f_{x-i,y-j}$$

- To apply blurring, a convolution mask in the form of a 5×5 Gaussian blur kernel is used:

$$\frac{1}{256} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

- This kernel performs the blur by computing a weighted average of the surrounding pixels, based on a distribution that approximates the Gaussian function. The mathematical formula of this distribution is:

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

- In addition, to handle the pixels at the edges of the image without causing dimension errors during convolution, the image is padded using a method called edge padding, which replicates the values of the border pixels to ensure that a full neighborhood is available for computation.
- At each pixel position, a weighted sum is computed over the surrounding 5×5 region (same size as the kernel), by multiplying each neighboring pixel value with the corresponding element in the kernel, and summing the results. This operation is performed simultaneously for all three RGB color channels.

To better understand the process, we will illustrate it using a 3x3 matrix (representing an image with a single color channel for simplicity; for an RGB image, the same steps would be applied separately to each channel) and a 5x5 Gaussian kernel provided earlier.

Original 3x3 matrix (Input image):

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Step 1: Padding

Since the kernel is a 5x5 matrix, the padding value will be 2. This is because when the center of the kernel is placed over the first pixel (0, 0) of the image, the edges of the kernel will overflow outside the image by 5 // 2 = 2 pixels. Therefore, we need to add a 2-pixel padding on each side to maintain the original image size after convolution.

After padding, the 3x3 input image becomes a 6x6 matrix:

$$A_{pad} = \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 & 2 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 3 & 3 & 3 & 4 & 4 & 4 \end{bmatrix}$$

Step 2: Perform convolution for each pixel

We start with pixel (0, 0). We take a 5×5 region from the top-left corner of the padded image:

$$X = \begin{bmatrix} 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 \\ 3 & 3 & 3 & 4 & 4 \\ 3 & 3 & 3 & 4 & 4 \end{bmatrix} \xrightarrow{convolution} value_{(0,0)} = \frac{31}{16} \Rightarrow B = \begin{bmatrix} \frac{31}{16} & \cdots \\ \cdots & \cdots \end{bmatrix}$$

Let matrix B be the resulting image after blurring.

Step 3: Repeat Step 2 for all remaining pixels

We repeat the same convolution process for the other pixels in the image, and finally obtain the full blurred matrix B.

$$B = \begin{bmatrix} \frac{31}{16} & \frac{37}{16} \\ \frac{43}{16} & \frac{49}{16} \end{bmatrix}$$

b. Image sharpening: sharpen_image()

- The process of sharpening an image is carried out in almost the same way as blurring. The key difference lies in the convolution kernel being used. Instead of applying a 5×5 Gaussian blur kernel, we use a sharpening kernel as follows:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

- Since the ideas, implementation steps, and a visual example have been thoroughly explained in the previous section on image blurring, they will not be repeated here.

5.2. Impletementation Details

a. Image blurring: blur_image()

- A 5×5 NumPy matrix is created to serve as the Gaussian kernel, with the data type set to *float32* to ensure precision during arithmetic operations such as addition and multiplication.
- The original image matrix is also converted to *float32*, and the .shape attribute is used to obtain the image's height and width.
- As explained in the implementation idea, since the kernel size is 5×5, the required padding is 5 // 2 = 2.
- The *blur* variable allows the blurring operation to be applied multiple times. Repeating the blur process enhances the overall blurring effect, as applying it just once tends to produce minimal visual change.
- Padding is performed using NumPy's *pad()* function with mode='edge', which replicates the values of the border pixels.
- Each element in the kernel is iterated over; the corresponding weight at position (i, j) is multiplied with the appropriate region in the padded image, and the result is accumulated into the *blurred* array (which has the same dimensions as the original image).
- After performing the blur operation *blur* times, *np.clip()* is used to limit pixel values within the range [0, 255] to prevent overflow caused

by accumulation. Finally, the result is cast to *uint8* so that the image displays correct colors when saved or shown.

b. Image sharpening: sharpen_image()

- Similar to the implementation in the *blur_image()* function, the *sharpen_image()* function also performs convolution between the original image and a kernel. However, instead of using a 5×5 Gaussian kernel, it uses a 3×3 sharpen kernel.
- Since the sharpening effect is often visibly strong after just one application, there is no need to repeat the operation multiple times as in the blurring process.

6. crop_center_image()

This function performs the operation cropping the center region of an image, with the input parameter *img_2d* – a 2D NumPy array representing the image.

6.1. Impletementation Idea

- To crop the central 1/4 portion of an image based on its size, we simply extract a region with: A height equal to half the original image height, and a width equal to half the original image width.
- To ensure that the cropped region is centered, we determine: The starting position: at one-fourth of the image's height and width. The ending position: at three-fourths of the image's height and width.
- Specifically, assuming the image has dimensions $H \times W$ the central region lies within the range:

$$y \in \left[\frac{H}{4}, \frac{3H}{4}\right], \quad x \in \left[\frac{W}{4}, \frac{3W}{4}\right]$$

- This ensures that the cropped image is centered and has an area equal to one-fourth of the original image.

6.2. Impletementation Details

- Use *.shape* to obtain the height and width of the original image
- Calculate the starting and ending coordinates for cropping along the vertical (y) and horizontal (x) axes, based on the idea explained above.
- Use NumPy slicing to extract the central region of the image by selecting: Rows from *start_*y to *end_y*, columns from *start_x* to *end_x*, while preserving all RGB color channels.
- This slicing operation allows the central portion of the image to be extracted efficiently without using loops, which significantly improves performance.

7. crop_circle_image() and crop_ellipse_image()

These two functions perform the operation of cropping the image using a circular mask and cropping the image using two intersecting ellipses, with the input parameter *img_2d* – a 2D NumPy array representing the image.

7.1. Impletementation Idea

a. Crop using a circular mask: crop_circle_image()

- To crop an image using a circular mask, we retain only the pixels that lie inside the circular region, while all pixels outside the circle are removed by setting their values to 0 (black).
- The process of determining whether a pixel lies inside the circle is based on the mathematical knowledge of the circle equation, commonly taught in high school geometry: A circle centered at point (a, b) with radius r has the equation:

$$(x - a)^2 + (y - b)^2 = r^2$$

- Each pixel in the image can be considered to have coordinates (x, y). A pixel lies on or inside the circle if:

$$(x - a)^2 + (y - b)^2 \leq r^2$$

- To ensure that the circular region fits entirely within the image boundaries, the radius r is chosen as half the length of the shorter side of the image (i.e., the minimum of the height and width).

b. Crop using two intersecting ellipses: crop_ellipse_image()

- Similar to cropping an image using a circular mask, this method involves retaining only the pixels that lie inside at least one of two rotated ellipses, while pixels that lie outside both ellipses are removed by setting their values to 0 (black).
- To determine whether a given pixel (x, y) lies within an ellipse, we use the general equation of a rotated ellipse, which is derived from analytical geometry. The general form of a rotated ellipse centered at the origin is:

$$\frac{(x\cos\theta + y\sin\theta)^2}{a^2} + \frac{(x\sin\theta - y\cos\theta)^2}{b^2} = 1$$

  Where: (x, y) are the coordinates of the pixel (with origin at the image center); a and b are the semi-major and semi-minor axes of the ellipse; θ is the angle of rotation relative to the x-axis
- To make this equation suitable for image processing, we normalize it using the image dimensions: Let $a = \frac{w}{2}$, $b = \frac{h}{2}$, where w is image width and h is height.

- The equation can be algebraically transformed into an equivalent normalized form [9]:

$$\left(\frac{2x}{w}\right)^2 + \left(\frac{2y}{h}\right)^2 + \frac{8\cos(\theta)xy}{hw} \leq \sin^2(\theta)$$

- This expression is then used to construct a Boolean mask, identifying which pixels fall within the rotated ellipse.
- In this implementation, we use two symmetric ellipses: The first ellipse is rotated by an angle $\theta_1 = 0.3\pi$. The second ellipse is rotated by $\theta_2 = \theta_1 - \pi$, which makes it symmetric across the center of the image.
- We apply both masks and retain all pixels that satisfy either ellipse condition. The final result preserves only the pixels within the union of the two ellipses, and sets the rest to black.
- Note on angle selection:

  - If the tilt angle is set as $\theta = k\pi$, the mask expression will evaluate to False for all pixels, since $sin^2(k\pi) = 0$. As a result, the entire image will be masked out and appear completely black after cropping.

  - If the angle is set as $\theta = \frac{1}{2}k\pi$, the two ellipses become identical and overlap completely, reducing the crop to a single ellipse, not two crossing ones.

7.2. Impletementation Details

a. Crop using a circular mask: crop_circle_image()

- Create a copy of the original image array *img_2d* as a NumPy array, and retrieve the image's height and width using *.shape*
- Compute the coordinates of the image center, with: *center_x = width // 2*, *center_y = height // 2*.
- The radius of the circle is set to half the length of the shorter side between height and width, ensuring that the circle fits entirely within the image boundaries.
- Generate a coordinate grid for each pixel in the image using NumPy's *ogrid()* function. This produces two arrays: *Y* with shape (h, 1), where each value represents a row index (from 0 to h - 1), repeated across columns. *X* with shape (1, w), where each value represents a column index (from 0 to w - 1), repeated down rows.
- Compute the squared distance from each pixel to the image center. NumPy automatically broadcasts the arrays so this calculation applies to every pixel in the image.

- Create a Boolean *mask* indicating whether each pixel lies inside the circle (True) or outside (False).
- Create an output image *circle_img* filled with zeros, and use *mask* to copy only the pixels that lie inside the circular region from the original image

b. Crop using two intersecting ellipses: crop_ellipse_image()

- Similar to the approach used for circular cropping, we first create a copy of the input image *img_2d* as a NumPy array and compute the center coordinates of the image: *center_x = width // 2*, *center_y = height // 2*.
- Define two tilt angles for the ellipses: $\theta_1 = 0.3\pi$ for the first ellipse, $\theta_2 = \theta_1 - \pi$ for the second ellipse
- Generate a coordinate grid for each pixel in the image using NumPy's *ogrid()* function, just like in the circular cropping method.
- Shift the coordinate system so that the origin is at the center of the image by applying: *X = X - center_x, Y = Y - center_y*
- Apply a normalized elliptical equation (derived from the rotated ellipse formula) to construct two Boolean masks: *mask1* identifies the region inside the first ellipse tilted by θ_1, *mask2* identifies the region inside the second ellipse tilted by θ_2. The general normalized equation used for each mask is provided above.
- Create an output image *ellipse_img* filled with zeros, and copy pixels from the original image that lie inside either of the two ellipses.

8. process_image()

This function acts as the central coordinator for the entire image processing pipeline. It takes two inputs: *img_2d* - a 2D NumPy array representing the original image, and a list of selected processing function numbers, specified in a predefined order. The function then sequentially applies each processing operation to the input image and displays the result after each step. If the list includes the value 0, it indicates that the user wants to save the processed images. In that case, all resulting images will be saved to new files, each named according to the function applied.

IV. References

1. Harris, C. R., et al. (2020). *NumPy*. https://numpy.org
2. Hunter, J. D., et al. (2024). *Matplotlib: Visualization with Python*. https://matplotlib.org
3. Clark, A. (2024). *Pillow (PIL Fork) Documentation* (v11.2.1). https://pillow.readthedocs.io
4. Pixabay. (n.d.). *Wilamowice Rapeseed Canola Fields - Free photo*. https://pixabay.com/photos/wilamowice-rapeseed-field-flowers-7585880
5. Pixabay. (n.d.). *AI generated, woman, young woman* [Illustration]. Pixabay. https://pixabay.com/illustrations/ai-generated-woman-young-woman-9562246/
6. Wikipedia contributors. (2024, March 17). *Kernel (image processing)*. Wikipedia. https://en.wikipedia.org/wiki/Kernel_(image_processing)
7. TutorialsPoint. (n.d.). *Mahotas – RGB to Sepia*. Retrieved July 26, 2025, from https://www.tutorialspoint.com/mahotas/mahotas_rgb_to_sepia.htm
8. TutorialsPoint. (n.d.). *Mahotas – RGB to Grayscale Conversion*. Retrieved July 26, 2025, from https://www.tutorialspoint.com/mahotas/mahotas_rgb_to_gray_conversion.htm
9. Stack Exchange. (2017, May 9). *Ellipse in a rectangle*. Mathematics Stack Exchange. https://math.stackexchange.com/questions/2308198/ellipse-in-a-rectangle
10. CodeLucky. (n.d.). *NumPy Image Processing: Basic Image Operations*. Retrieved July 26, 2025, from https://codelucky.com/numpy-image-processing/
11. Khanhnhan1512. (n.d.). *Image-Processing: This is a project of Applied Maths in HCMUS* [GitHub repository]. GitHub. https://github.com/khanhnhan1512/Image-Processing

V. Acknowledgement

To complete this project, I would like to express my sincere gratitude to my friends, instructors, and AI tools that supported me throughout the implementation process:

- **Tran Cong Minh** – a student from the Applied Mathematics and Statistics class 23CLC05, Student ID 23127007 – who helped me implement the crop_ellipse_image() function and showed me how to use global variables in the read_img() and process_image() functions in order to save images with a new path based on the original file path. Additionally, the *save_img()* function from the previous project, which was also implemented with help, has also been reused in this project.
- **ChatGPT** – an AI tool that guided me in structuring this report and correcting grammatical errors throughout. It also assisted me in improving several parts of the code, such as optimizing the image blurring function, helping me understand convolution in image processing, and providing deeper insights into powerful NumPy tools such as slicing and broadcasting. Additionally, it also helped me format the references section according to the APA 7th edition citation style.
- Finally, I would like to express my sincere thanks to the instructors of the Applied Mathematics and Statistics course: **Mr. Vu Quoc Hoang, Ms. Tran Thi Thao Nhi,** and **Mr. Nguyen Ngoc Toan**, who provided me with valuable knowledge that I was able to apply to this project.