NATIONAL UNIVERSITY, HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY


# Image Processing

## Implemented using Python and NumPy

### Course: Applied Mathematics and Statistics

*Student:*

23127136 - Lê Nguyễn Nhật Trường

*Teachers:*

Vũ Quốc Hoàng

Trần Thị Thảo Nhi

Nguyễn Văn Quang Huy

July 27, 2025

# Contents

# List of Figures



# List of Tables

# 1. Introduction

## 1.1. Image Processing

- **Image Processing** is the process of applying various processing functions, resulting in many possible manipulations of the image.
- Since each image is a matrix of pixels and each pixel contains the three color channels Red, Green, and Blue (RGB), image processing can be considered as a matrix manipulation problem.
- In this project, I will use **Python** with mainly the **NumPy** library to implement the processing functions. Some other libraries such as **Matplotlib**, **Colorsys** and **Pillow** will also be used for mundane tasks like reading, displaying, saving images, and converting color spaces.
- Below is an example of an image after processing. The *original image* here will also be used throughout the project to demonstrate the results of the processing functions.

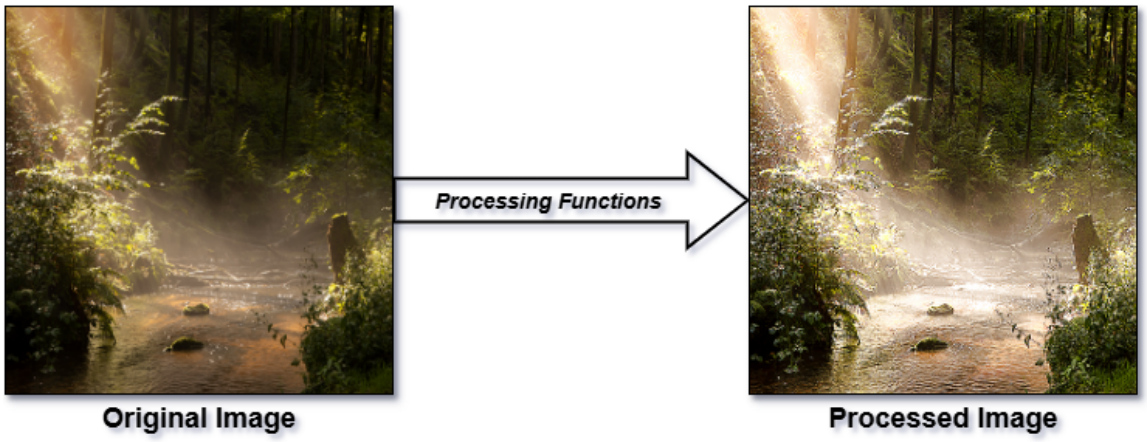



Figure 1.1: Example of an image after processing

## 1.2. Processing Functions

- The processing functions that are implemented include:
  - **Brighten Image**: Increase the brightness of the image.
  - **Increase Contrast**: Enhance the contrast of the image.
  - **Flip Image**: Flip the image horizontally (mirrored) or vertically.
  - **Convert to Grayscale/Sepia**: Convert the image to grayscale or sepia tone.
  - **Blur and Sharpen Image**: Apply blurring or sharpening effects to the image.
  - **Crop Image by Size**: Crop to one fourth of the image from the center.
  - **Crop Image by Frame**: Crop the image using a circular frame, or a double ellipse frame.
- Additionally, a **saving function** is implemented in the process_image() function, which I will explain in detail later.
- When running the program, each function will be labeled with an integer from 1 to 7, and the user can choose which function to apply to the image.
- Some functions will have multiple results, meaning both operations will be applied to the image, and all results will be returned and saved in the same folder. These functions include: **Flip Image**, **Convert to Grayscale/Sepia**, **Blur and Sharpen Image** and **Crop Image by Frame**.
- In the next sections, I will describe the implementation of the helper functions and the processing functions in detail, with examples of their results.
- ***Note**: To test these functions, I will use royalty-free images sourced from pixabay*[1].

# 2. Helper Functions

## 2.1. Read and Show Image

- The function `read_img()` reads an image from a given path and returns it as a NumPy array. Using the **Pillow** library, it opens the image file and converts it to RGB format. Then, I convert the image to a NumPy array and return it. Furthermore, the function also save the image path to a global variable `saved_image_path`. This is useful for later saving the processed image in the `process_image()` function.

- The function `show_img()` will simply print the image size (height $\times$ width) and display the image using the `imshow()` function from **Matplotlib**.

## 2.2. Color Space Conversion

- The function `convert_rgb_to_hsl()` converts an NumPy array of RGB values to HSL (Hue, Saturation, Lightness) color space. It uses the function `colorsys.rgb_to_hls()` from the **Colorsys** library to perform the conversion. The function returns a NumPy array of HSL values, each value using a float type in the range [0, 1].

- Similarly, the function `convert_hsl_to_rgb()` converts an NumPy array of HSL values back to RGB color space. It uses the function `colorsys.hls_to_rgb()` from the **Colorsys** library to perform the conversion. The function returns a NumPy array of RGB values, each value using a uint8 type in the range [0, 255].

- These functions are useful for processing images, especially when changing the contrast or saturation of an image. The HSL color space allows for more intuitive manipulation of these properties compared to RGB.

## 2.3. Additional Handling: Consecutive Processing

- The function `consecutive_process_image()` is a wrapper function that allows for consecutive processing of an image using a list of functions. It takes an image and a list of function numbers as input, applies the corresponding functions in order, and returns the processed image.
- For any branching processing function, such as **Blur and Sharpen** or **Crop by Frame**, the function will branches into multiple images, each processed by a different function, then applies the next function in the list to each of these images separately. Additionally, the function will also save the processed images with the corresponding suffixes to the original image path.
- Originally, this function was my first implementation of the `process_image()` function. After the first week, I changed the implementation to a more simpler version, which is currently used in the `process_image()` function. However, I still keep this function for future use, as it is more flexible and allows for more complex processing sequences.
- Here is an example image of how this function will work:

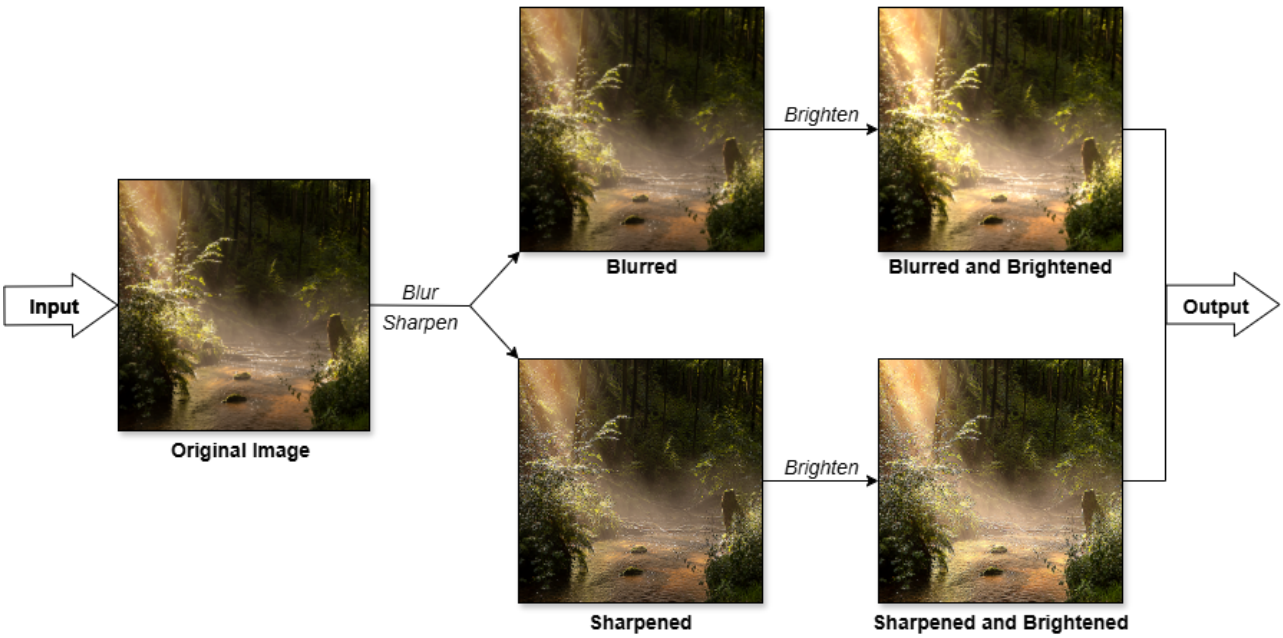



Figure 2.1: Running [*Blur and Sharpen, Brighten*] on an image

# 3. Process Image Function

## 3.1. Main Idea

- The `process_image` function is an important function in this project. It will receive a 2D NumPy array representing the image and a list of functions to apply to that image.
- All functions in the list will be applied separately to the original image, and the results will be saved in a list of 2D NumPy arrays.
  - **Input**: img_2d (2D NumPy array), func (list of function numbers)
  - **Output**: results (list of 2D NumPy arrays representing processed images)
  - **Example Image**: Here is an example of how the function run:

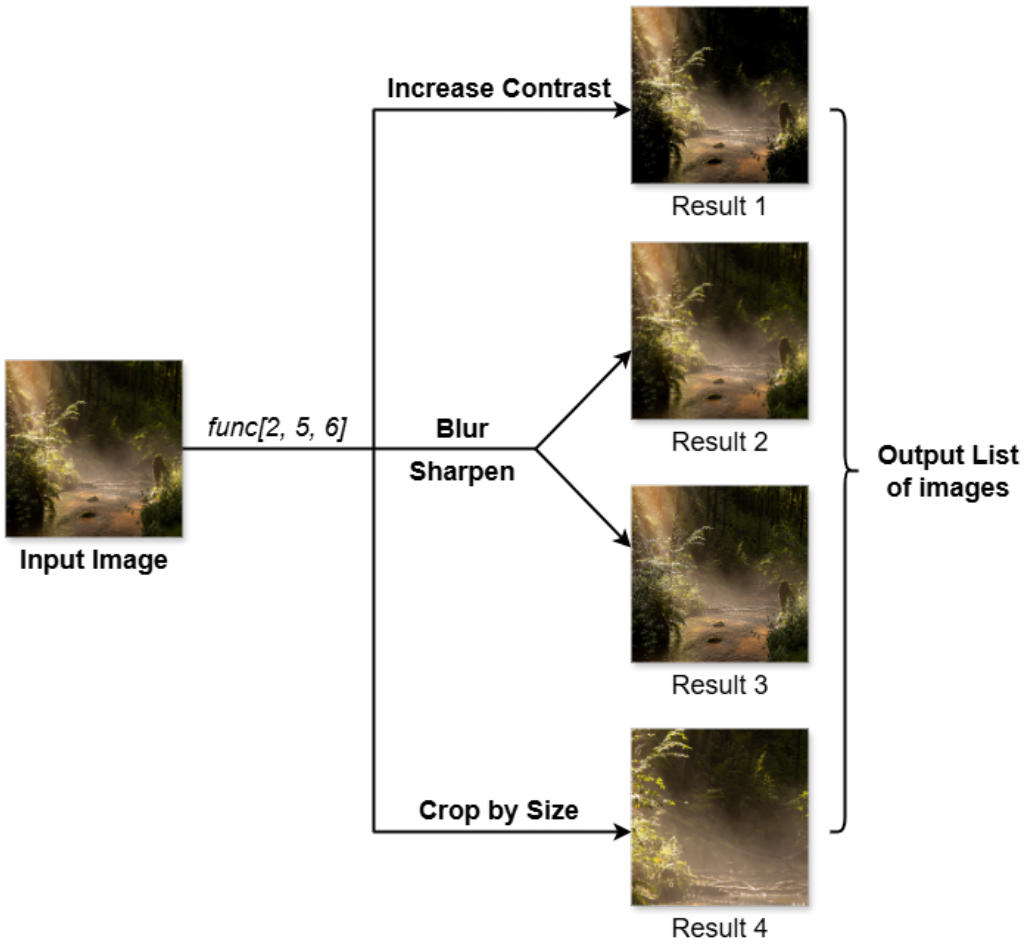



Figure 3.1: Example of the `process_image` function run

## 3.2. Saving Processed Images

- Inside the `process_image` function, if 0 is in the func list, the function will save the processed images with the suffix corresponding to the processing function applied.
- If 0 is **NOT** in the function list, the function will not save any images and will only return the processed images.
- The saved images will be named according to the original image name with a suffix indicating the processing function applied, such as `cat_blur.png` for a blurred image.
- The original image name will be automatically saved to a global variable when running the `read_img()` function, so there is no need to use an additional parameter for the image path in the `process_image` function.
- Furthermore, the function will save all processed images in the same directory as the original image, which should also be the root directory of the files.
- Since there are 7 processing functions, and some have multiple results, suffixes will be mapped to the function numbers as follows:
  - 1: _brighten
  - 2: _contrast
  - 3: (_verticalFlip, _mirroredFlip)
  - 4: (_grayscale, _sepia)
  - 5: (_blur, _sharpen)
  - 6: _cropped
  - 7: (_circularCrop, _doubleEllipseCrop)

## 3.3. Step-by-Step Implementation

1) **Function Entry and Error Checking:** The function receives the 2D NumPy array and a list of function numbers. It first checks if the image is missing, raising an error if necessary. This image path is saved automatically to a global variable when running the `read_img()` function.

2) **Mapping Functions and Suffixes:** It creates dictionaries to map function numbers to actual processing functions and to suffixes for appending suffixes to saved images.

3) **Preparing the Function List:** If 0 is in the function list, it sets a flag to save images. It removes 0 from the list and checks if the list is empty, returning the original image if so.

4) **Handling Empty Path:** If the save flag is set but the image path is empty, it will print out an error message and use a default name for the images: `"image.png"`.

5) **List Initialization:** The function initializes 2 lists: one for all processed images and one for each image's path. This will be used to save the processed images with the correct suffix.

6) **Main Processing Loop:** For each function number in the list, the function applies the corresponding processing function to the original image. It checks if the function is valid and applies it, adding the processed images to the results list.

7) **Conversion to List:** To make sure the variables are in the correct format, it converts them to lists if they are single instances. This happens with the processed results and the suffixes.

8) **Saving Images:** If the save flag is set, the function saves each processed image with the correct filename and suffix. Each image will have the same index in the processed images list and the saved image path list.

9) **Returning Results:** Finally, the function will return the list of processed images, which can be used for further processing or display.

# 4. Processing Functions

## 4.1. Function 1: Brighten Image

- **Description**: This function increases the brightness of an image by scaling up the pixel values, useful for enhancing visibility in dark images.
- **Implementation**:
  - The function takes a 2D NumPy array representing the image and a multiplier factor. It multiplies each pixel value by this factor, ensuring that values do not exceed the maximum allowed by clipping to the range `[0, 255]`.
  - The multiplier is set to **1.2** by default, because it provides a noticeable increase in brightness without overexposing the image for each repetition.
  - By using NumPy's vectorized operations, the function efficiently scales the pixel values, which is faster than iterating through each pixel.
- **Example Result**: Here is a list of images side-by-side after applying the brighten function for different numbers of times:

(a) Original Image (b) Brightened Once (c) Brightened Thrice (d) Brightened 5 Times

Figure 4.1: Brightening Effect on Image

- As we can see, by one time, the image has already appeared brighter, and by five times, the image is too bright, losing details from the original image.

## 4.2. Function 2: Increase Contrast

- **Description**: This function enhances the contrast of an image by applying a linear transformation to the pixel values, making dark areas darker and light areas lighter.
- **Implementation**:
  - The function first converts the NumPy array from RGB to HSL color space, which helps separate the Lightness factor from the color information.
  - It then scales out the Lightness values (L) using the formula:

$$L_{middle} = \frac{\max(L) + \min(L)}{2}$$

$$L_{new} = multiplier \times (L - L_{middle}) + L_{middle}$$

- This formula is designed to increase the contrast by adjusting the Lightness values around the middle point of the lightness range.
- The multiplier is set to **1.2** by default, which provides a noticeable contrast enhancement without making the image too extreme.
- **Example Result**: Here is a list of images side-by-side after applying the increase contrast function for different numbers of times:

(a) Original Image (b) Contrast Once (c) Contrast Thrice (d) Contrast 5 Times

Figure 4.2: Contrast Enhancement Effect on Image

- As we can see, the contrast enhancement makes the dark areas darker and the light areas lighter. By 5 times, the darker areas are almost black, while the lighter areas are quite bright.

## 4.3. Function 3: Flip Image

- **Description**: This function flips an image horizontally and vertically, resulting in 2 different orientations of the original image.
- **Implementation**:
  - The function simply takes a 2D NumPy array representing the image and flips it using NumPy's slicing capabilities.
  - It returns two versions of the image: one flipped vertically and one mirrored.
  - Both flipped images are created using the following NumPy slicing: `::-1`, which will takes all elements of that axis but in reverse order.
  - Therefore, **[::-1]** will reverse the order of the rows (vertical flip) and **[:, ::-1]** will reverse the order of the columns (horizontal flip).
- **Example Result**: Here are the flipped images alongside the original image:

(a) Original Image (b) Flipped Vertically (c) Flipped Horizontally (d) Flipped Diagonally

Figure 4.3: Flipping Effect on Image

- As we can see, the vertical flip and horizontal flip create mirror images of the original image, while the diagonal flip combines both effects.

## 4.4. Function 4: Convert to Grayscale and Sepia

- **Description**: This function converts an image to grayscale and sepia tones separately, resulting in two different processed images.
- **Implementation**:
  - **Grayscale Conversion**: The function calculates the average of the RGB values for each pixel and sets all three color channels to this average value, effectively removing color while retaining details from the original image.
  - **Sepia Conversion**: For this conversion, the function uses a sepia filter matrix:

$$\texttt{Sepia Filter} = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix}$$

  - *The sepia filter matrix was adapted from a method presented in MSDN Magazine.*[1]
  - To apply the sepia effect, the function uses NumPy's dot product to multiply the RGB values of each pixel by the sepia filter matrix, resulting in a new set of RGB values that is then clipped to the range `[0, 255]` to avoid overflow.
- **Example Result**: Here is the grayscale and sepia images alongside the original image:

(a) Original Image (b) Grayscale Image (c) Sepia Image

Figure 4.4: Grayscale and Sepia Effect on Image

[1] Only the matrix values were adapted from [2]; no other parts of the original implementation were referenced.

- As we can see, the grayscale image has lost colors while still retaining many details such as the leaves texture, the light beams, or the depth of the forest. On the other hand, the sepia image gives a warm, vintage look with an overall yellowish-brown tint.
- These effects are quite useful for creating artistic representations of images, radiating new emotions and atmospheres, or even for preparing images for further processing in computer vision tasks.

## 4.5. Function 5: Blur and Sharpen Image

- **Description**: This function applies a Gaussian blur to an image, which smooths out noise and details, and a sharpening filter to a separate image that enhances edges and fine details.
- **Implementation**:
  - The Gaussian blur and sharpen masks were implemented using standard convolution kernels.[2]

$$\texttt{Gaussian Blur Mask} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\texttt{Sharpen Mask} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

  - The Gaussian blur mask is a 3x3 matrix that averages the pixel values in a neighborhood, effectively smoothing the image.
  - The sharpen mask is also a 3x3 matrix that enhances the edges by subtracting the average of the surrounding pixels from the center pixel, making it stand out more.

[2]Only the 3x3 matrix values and method descriptions for Gaussian blur and sharpening were referenced from [3].

  - To apply these masks, the function loops through each pixel in the image and performs a convolution operation with the respective mask, calculating the new pixel values based on the surrounding pixels.
  - The convolution is done by looping through pixels around the target pixel, multiplying each surrounding pixel value by the corresponding mask value, and summing these products to get the new pixel value.
  - The resulting pixel values are then clipped to the range `[0, 255]` to ensure valid RGB values.

- **Example Result**: Here is some blurred and sharpened images alongside the original image:

(a) Original Image (b) Blurred Image (c) Sharpened Image

Figure 4.5: Blurring and Sharpening Effect on Image

(a) Blurred Twice (b) Blurred Thrice (c) Sharpened Twice (d) Sharpened Thrice

Figure 4.6: Extra Blurring and Sharpening Effect on Image

- As we can see, the blurred image has a smooth appearance with less detail, while the sharpened image has enhanced edges and fine details.

- The blur effect get stronger with each application, while the sharpen effect get quite extreme. An interesting observation is that the sharpened twice image has a bright background compared to the original image, making each bush or tree very easy to identify, while the sharpened thrice image gets too extreme and completely ruins the entire image.

- Since there is 4 nested loops in the implementation, the performance of this function becomes worse with larger images, here is a table showing the time taken to process images of different sizes:

| **Image Size (HxW)** | **Average Runtime (s)** |
|---|---|
| 128x128 | 0.5569 |
| 256x256 | 2.2480 |
| **512x512** | **9.2788** |
| 1024x1024 | 37.8762 |
| 2048x2048 | 153.1936 |

Table 4.1: Processing Time for Blur and Sharpen Functions on Different Image Sizes

- The data was measured using a randomly generated NumPy array of the specified size (h x w x 3), and the average runtime was calculated over 10 runs for each size.

- As we can see, the processing time increases significantly with larger images, especially for the 1024x1024 and 2048x2048 images. However, for the **512x512 image**, it is still acceptable as it takes less than **15 seconds** to process.

- A trade-off for this exponential increase in runtime is that the implementation remains simple and easy to understand, since it does not use any array padding or resizing in NumPy.

## 4.6. Function 6: Crop Image by Size

- **Description**: This function crops an image to a specified size, focusing on the central region. It is useful for zooming in on a particular central area of the image or reducing the image size for further processing.
- **Implementation**:
  - The function takes a 2D NumPy array representing the image and a size factor (default is 0.25 for quarter-size cropping).
  - It calculates the new height and width by multiplying the original dimensions by the size factor. This will help preserve the original aspect ratio.
  - The crop is centered: it calculates the top-left and bottom-right coordinates so that the cropped region is in the center of the original image.
  - The function slices the NumPy array accordingly and returns the cropped image.
  - If the size factor results in a size larger than the original, the function simply returns the original image.
- **Example Result**: Here is an example of cropping an image to a smaller size, using a new image with a different aspect ratio:

(a) Original Image (b) Cropped 1/2 Size (c) Cropped 1/4 Size (d) Cropped 1/8 Size

Figure 4.7: Cropping Effect on Image

- As we can see, the cropped image retains the central part of the original image, focusing on the main subject while removing the surrounding areas and maintaining the original aspect ratio.

## 4.7. Function 7: Crop Image by Frame

- **Description**: This function creates artistic crops of the image using geometric masks: a circular crop and a double-ellipse crop.
- **Implementation**:
  - The function takes a 2D NumPy array representing the image.
  - It creates a circular mask centered in the image, with a radius equal to half of the smaller dimension. Pixels outside the circle are set to zero.
  - For the double-ellipse crop, it generates two ellipses rotated at $+45°$ and $-45°$, both are centered and sized to fit within the image (similarly to the circular mask).
  - The function uses NumPy's meshgrid to create a grid of coordinates, which allows it to calculate the distance from the center for each pixel. To create the masks, it uses the following equations:

$$\texttt{Circle Mask:} \boxed{\sqrt{(x - c_x)^2 + (y - c_y)^2} \leq radius}$$

$$\texttt{Rotated Ellipse Mask:}$$

$$\boxed{\frac{[(x - c_x) \times \cos(\theta) + (y - c_y) \times \sin(\theta)]^2}{a^2} + \frac{[(x - c_x) \times \sin(\theta) - (y - c_y) \times \cos(\theta)]^2}{b^2} \leq 1}$$

    + $c_x$, $c_y$ are the center coordinates of the image.
    + $radius$ is half of the smaller dimension of the image for the circular mask.
    + $a$ is set to $0.87 \times \frac{\sqrt{2}}{2}$ times the smaller dimension (i.e., proportional to the diagonal of the largest inscribed square)
    + $b$ is set to one third of the smaller dimension of the image.
    + $\theta$ is the rotation angle, set to $45°$ for one ellipse and $-45°$ for the other.

  - The rotated ellipse equation used for this implementation was adapted from a Math Stack Exchange answer.[3]
  - The circular mask is created by applying the circular equation, while the double-ellipse mask is created by combining two rotated ellipse equations (each rotated 45° and −45°). Pixels inside the shapes are set to one, and pixels outside are set to zero
  - Finally, both masks are applied to the image using NumPy broadcasting, resulting in two output images: one circularly cropped and one double-ellipse cropped.

- **Example Result**: Here are the circular and double-ellipse cropped images alongside the original image, with a vertical and a horizontal image for comparison:

(a) A Vertical Image (b) Circular Crop (c) Double-Ellipse Crop

Figure 4.8: Frame Cropping Effect on A Vertical Image

(a) A Horizontal Image (b) Circular Crop (c) Double-Ellipse Crop

Figure 4.9: Frame Cropping Effect on A Horizontal Image

[3]Only the formula provided by user **andikat dennis** in the top answer of [4] was referenced.

(a) A Square Image (b) Circular Crop (c) Double-Ellipse Crop

Figure 4.10: Frame Cropping Effect on A Square Image

- As we can see, both of the cropping frames fit perfectly within the original image, regardless of its aspect ratio. Both of the cropping frames retain the central part of the image, focusing on the main subject while removing (blacking out) the surrounding areas.

- For the circular crop, the frame fits easily with only scaling the radius to half of the smaller dimension, while for the double-ellipse crop, the frame is harder to fit, since it requires calculating the $a$ and $b$ values based on the smaller dimension. And with some testing, I have found that the values $a = 0.87 \times \frac{\sqrt{2}}{2} \times$ smaller dimension and $b = \frac{1}{3} \times$ smaller dimension work well, as it will fit the frame within a square that is inscribed within the original image.

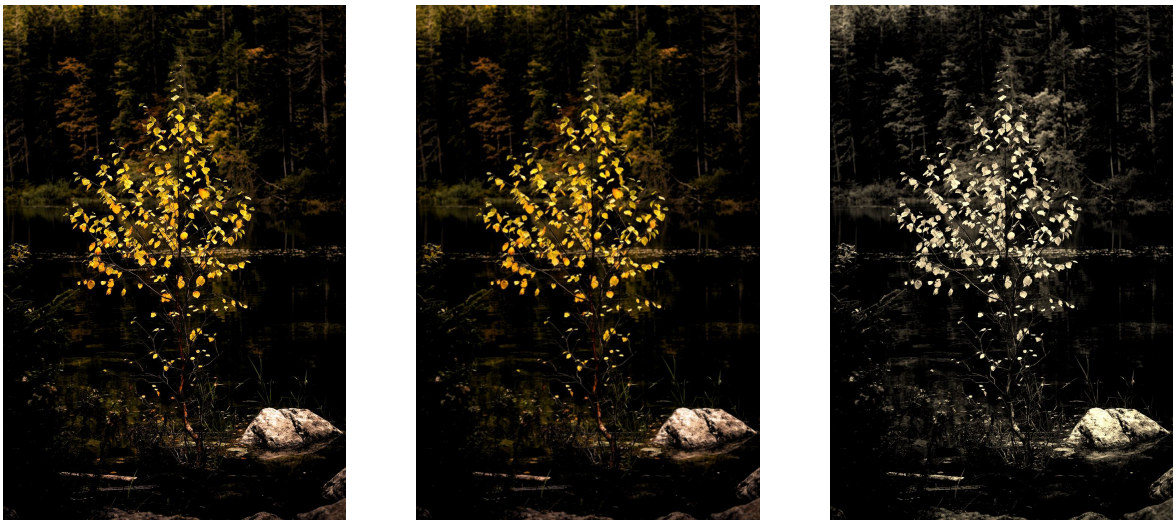
# 5. Additional Results

- In this section, I will present additional results from the image processing functions implemented in the project.
- The results will be achieved by applying various processing functions to the original image, which can be done effectively using the `consecutive_process_image()` function from 2.3.
- The results will show how we can create many different images by applying multiple processing steps. Some simple functions like **Brighten** and **Increase Contrast** are extremely useful and can be applied multiple times and in different orders to achieve different effects.
- Here are the original images, then the results of applying the processing functions:

(a) Original Image: Forest

(b) Original Image: Leaf

Result 1 Result 2 Result 3



Result 4 Result 5 Result 6

Figure 5.2: Additional results of applying processing functions

- The results shown above are achieved by applying the following processing functions in sequence:
  - **Result 1**: Brighten, Increase Contrast, then Brighten again.
  - **Result 2, 3**: Result 1, then convert to Grayscale and Sepia.
  - **Result 4**: Brighten twice, then Increase Contrast.
  - **Result 5, 6**: Result 4, then Blur the image and convert to Sepia.

# References


[1] Pixabay contributors, "Pixabay – free images and videos," 2025, used for royalty-free images in this project. Available at: https://pixabay.com [Accessed: July 2025].

[2] J. Prosise, ".net matters: Sepia tone, stringlogicalcomparer, and more," *MSDN Magazine*, Jan. 2005, available at: https://learn.microsoft.com/en-us/archive/msdn-magazine/2005/january/net-matters-sepia-tone-stringlogicalcomparer-and-more [Accessed: July 2025].

[3] Wikipedia contributors, "Kernel (image processing) — wikipedia, the free encyclopedia," 2024, available at: https://en.wikipedia.org/wiki/Kernel_(image_processing) [Accessed: July 2025].

[4] andikat dennis, "What is the general equation of the ellipse that is not in the origin and rotated?" Math Stack Exchange, 2013, answer to question ID 426150. Available at: https://math.stackexchange.com/q/426150 [Accessed: July 2025].