

UNIVERSITY OF SCIENCE - VIETNAM NATIONAL UNIVERSITY OF HCM

Faculty of Information Technology

INTRODUCTION TO INTELLIGENCE ARTIFICIAL - 21CLC07

*Report*  
**PROJECT 02**  
**IMAGE PROCESSING**



**LECTURER**

Ngô Đình Hy  
Nguyễn Văn Quang Huy  
Trần Hà Sơn  
Nguyễn Đình Thúc

**MEMBER**

Full Name : Phạm Hồng Gia Bảo  
MSSV : 21127014

2022 - 2023

# TABLE OF CONTENTS

<b>TABLE OF CONTENTS .....</b>	<b>2</b>
<b>I. PERSONAL INFORMATION.....</b>	<b>3</b>
<b>II. ACHEIVEMENT .....</b>	<b>3</b>
<b>III. UAGE – HOW TO USE PROGRAM.....</b>	<b>3</b>
<b>IV. ALGORITHMS AND IMPLEMENTATION.....</b>	<b>4</b>
1. Adjust brightness of Image .....	4
2. Adjust contrast of Image .....	5
3. Flip Image (Vertical/Horizontal) .....	6
4. Convert from RBG to Grayscale/Sepia.....	6
5. Blur and Sharpen Image.....	7
6. Crop Image to Size (Crop in Center) .....	11
7. Crop Image in a Circular Frame.....	12
8. Advanced: Crop Image in frame is 2 diagonal ellipses.....	13
9. Main Function .....	14
<b>V. RESULT .....</b>	<b>15</b>
<b>VI. REFERNCE.....</b>	<b>19</b>

## I. PERSONAL INFORMATION

<b>Student's Name</b>	Phạm Hồng Gia Bảo
<b>Student's ID</b>	21127014
<b>Class</b>	21CLC07
<b>Email</b>	phgbao21@clc.fitus.edu.vn

## II. ACHEIVEMENT

No.	Task Requirements	Completed
1	Adjust brightness of Image	100%
2	Adjust contrast of Image	100%
3	Flip Image (Vertical/Horizontal)	100%
4	Convert from RBG to Grayscale/Sepia	100%
5	Blur and Sharpen Image	100%
6	Crop Image to Size (Crop in Center)	100%
7	Crop Image in a Circular Frame	100%
8	<b>Advanced:</b> Crop Image in frame is 2 diagonal ellipses	100%
9	Main Function	100%

## III. UAGE – HOW TO USE PROGRAM

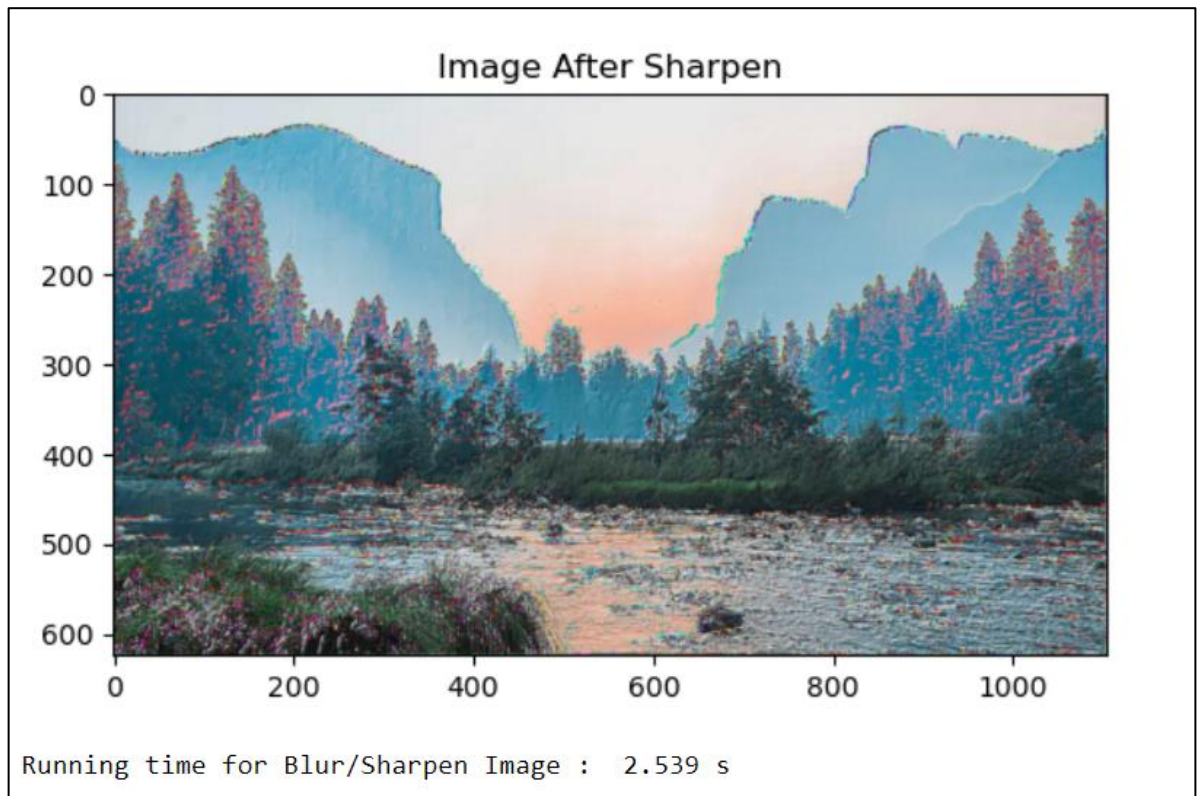
- **Step 1:** Input the name of the image file.

Enter image name:

- **Step 2:** Enter the option of the task you want to do.

```
***** MAIN MENU *****
#0  Do all tasks
#1  Adjust Brightness
#2  Adjust Contrast
#3  Flip Image (Vertical/Horizontal)
#4  Convert from RBG to Grayscale/Sepia
#5  Blur and Sharpen Image
#6  Crop Image to Size (Crop in Center)
#7  Crop Image in a Circular Frame
#8  Advanced: Crop Image in frame is 2 diagonal ellipses
*****
Enter option : 
```

- **Step 3:** View the result and check the output file.



## IV. ALGORITHMS AND IMPLEMENTATION

### 1. Adjust brightness of Image

#### a. Algorithm

- Adjusting the brightness means, either increasing the pixel value evenly across all channels for the entire image to increase the brightness or decreasing the pixel value evenly across all channels for the entire image to decrease the brightness.
- **Pseudocode**
  - Load the image into memory.
  - Convert the image to an array representation, so that pixel values range from 0 (black) to 255 (white).
  - Multiply each pixel value by a scaling factor to adjust the brightness.
  - Ensure that the pixel values are still within the valid range of [0,255]. If any values are outside this range, clip them to the nearest valid value.
  - Save the adjusted image to disk.

#### b. Implementation

```
def changeBrightness(img, name_img):
    brightValue = 150
    result = np.uint8(np.clip(img + np.array([brightValue], dtype=np.int16), 0, 255))
    plt.imshow(result)
    Image.fromarray(result).save(name_img.split('.')[0] + '_brightness_at_' + str(brightValue) + '.png')
```

- Firstly, we define a variable called **brightValue = 150** (brightValue in range of  $[-255, 255]$ ) that represents the amount of brightness to add to the image.
- Then, we calculate the new brightness-adjusted image by adding **brightValue** to the input image array, converts the result to an unsigned 8-bit integer using **np.uint8**, and clips the result to ensure that all pixel values are in range  $[0, 255]$  using **np.clip** from “numby” library.
- Then save it to new .png file.

## 2. Adjust contrast of Image

### a. Algorithm

- Contrast can simply be explained as the **difference between maximum and minimum pixel intensity** in an image. To change the contrast of an image we just need to change the value of the max and min intensity pixel. But if we are just changing the value of two pixels there won't be any difference in the two images.
- To calculate a contrast correction factor which is given by the following formula:  $F = 259 * (255 + C) / 255 * (259 - C)$  (in formulation taken from the article in the Reference)
- **Pseudocode**
  - Load the image.
  - Convert the image to an array representation, so that pixel values range from black to white which is in range of  $[0, 255]$ .
  - Compute the **mean and standard deviation of the pixel values** in the image.
  - Subtract the mean from each pixel value, and then divide by the standard deviation. This scales the pixel values so that they have zero mean and unit variance. Multiply each pixel value by a scaling factor to adjust the contrast.
  - Ensure that the pixel values are still within the valid range of  $[0, 255]$ . If any values are outside this range, clip them to the nearest valid value.
  - Save the adjusted image to .png file.

### b. Implementation

```
def adjust_contrast(img, name_img):
    contrast = 200
    contrast = np.clip(float(contrast), -255, 255)
    factor = (259 * (contrast + 255)) / (255 * (259 - contrast))
    result = np.uint8(np.clip(factor * (img.astype(float) - 128) + 128, 0, 255))
    plt.imshow(result)
    output_file = name_img.split('.')[0] + '_contrast_at_' + str(contrast) + '.png'
    Image.fromarray(result).save(output_file)
```

- Firstly, we set the **contrast\_value** which can be adjusted to increase or decrease the contrast of the image.
- Then, we use **np.clip()** function to ensure that contrast value is within the valid range of  $[-255, 255]$ .
- We calculate the **scaling factor** for adjusting the contrast of the image. The formula used here is explained above, and it maps the pixel values of the input image to a new range that increases or decreases the contrast. Then, the scaling factor is computed using the contrast value.
- After that, we apply the **contrast adjustment** to the input array image.
- Save the adjusted image to .png file by the name formatting in request.

### 3. Flip Image (Vertical/Horizontal)

#### a. Algorithm

- For flipping horizontally, iterate over the rows of the image matrix and swap the elements of each row with their corresponding elements in the opposite column.
- For flipping vertically, iterate over the columns of the image matrix and swap the elements of each column with their corresponding elements in the opposite row.

#### b. Implementation

- Use the **numpy.flipud()** to flip the entries in each column in the up/down direction. Rows are preserved but appear in a different order than before.
- Use the **numpy.fliplr()** to flip the entries in each row in the left/right direction. Columns are preserved but appear in a different order than before.

```
def flipImg(image, name_img):
    print('#1 : Vertical')
    print('#2 : Horizontal')
    direction = input("Enter flip direction: ")
    if direction == '1':
        result = np.flipud(image)
    elif direction == '2':
        result = np.fliplr(image)
    plt.imshow(result)
    output_file = name_img.split('.')[0] + '_flip_' + direction + '.png'
    Image.fromarray(result).save(output_file)
```

### 4. Convert from RGB to Grayscale/Sepia

#### a. Algorithm

- Converting an image to grayscale involves converting each pixel of the image from a color (RGB) value to a grayscale/sepia value. The main difference between Grayscale and Sepia is matrix value. While Grayscale is 1D-array  $[0.3, 0.59, 0.11]$ , Sepia matrix is a  $3 \times 3$  Matrix  $\begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix}$ . To do this, we apply the luminosity method which calculates the grayscale/sepia value of each

pixel based on its RGB values, weighted by the perceived brightness of each color channel.

- Here's the algorithm to convert an image to grayscale/sepia using the luminosity method:
  - Load the image and get its dimensions (width and height).
  - Create a new 2D array of the same dimensions as the image, initialized with zeros.
  - Compute the Grayscale/Sepia image by taking the dot product of the input image with the Grayscale/Sepia matrix.
  - Save the new 2D array as an image file in Grayscale/Sepia format.

#### b. Implementation

- Define the transformation Grayscale/Sepia matrix for converting RGB to Grayscale/Sepia.
- Firstly, we define the weights for each color channel as a 1D NumPy array `gray_matrix/sepia_matrix`, which corresponds to the luminosity method.
- The function then computes the grayscale image by taking the dot product of the input image with the weights using NumPy's `dot` function. The resulting grayscale image is a 2D NumPy array of unsigned 8-bit integers (**`np.uint8`**) representing the grayscale/sepia values of each pixel.
- The function then displays the grayscale/sepia image using matplotlib's **`imshow()`** function to specify that the image should be displayed.
- Finally, the function saves the grayscale/sepia image as a PNG file using the PIL library's **`Image.fromarray()`** and **`Image.save()`** functions.

```
def grayScale(img, name_img):
    gray_matrix = np.array([0.3, 0.59, 0.11])
    result = np.uint8(np.dot(img[..., :3], gray_matrix))
    plt.imshow(result, cmap='gray')
    Image.fromarray(result).save(name_img.split('.')[0] + '_grayscale' + '.png', cmap='gray')
```

```
def sepiaColor(image, name_img):
    sepia_matrix = np.array([[0.393, 0.769, 0.189],
                             [0.349, 0.686, 0.168],
                             [0.272, 0.534, 0.131]])
    sepia_image = np.dot(image[..., :3], sepia_matrix.T)
    sepia_image = np.clip(sepia_image, 0, 255)
    sepia_image = sepia_image.astype(np.uint8)
    plt.imshow(sepia_image)
    Image.fromarray(sepia_image).save(name_img.split('.')[0] + '_sepia' + '.png')
```

## 5. Blur and Sharpen Image

#### a. Blur Image

- *`def convolve_layer(layer, kernel):`* performs the convolution operation between a layer and a kernel. It utilizes NumPy's **`as_strided()`** function to create submatrices from the layer based on the kernel size and strides. The **`einsum()`** is then used to perform the actual convolution operation.



```
def convolve_layer(layer, kernel):
    view = kernel.shape + tuple(np.subtract(layer.shape, kernel.shape) + 1)
    submatrices = np.lib.stride_tricks.as_strided(layer, shape = view, strides = layer.strides * 2)
    return np.einsum('ij,ijkl->kl', kernel, submatrices)
```

- **def blur\_Img(image, name\_img):** applies a Gaussian blur to an image using the convolve\_layer()

- It defines the **kernel\_size** and calculates the standard deviation (sigma) based on the kernel size.

```
kernel_size = 40
sigma = (kernel_size - 1)/6
```

- It generates a 1D Gaussian kernel by creating an array of values from  $-(\text{kernel\_size} // 2)$  to  $(\text{kernel\_size} // 2)$  and applying the Gaussian function to each value.

```
# Gaussian_kernel
kernel_1d = np.linspace(-(kernel_size // 2), kernel_size // 2, num=kernel_size)
kernel_1d = np.array(1 / (np.sqrt(2 * np.pi) * sigma) * (np.exp(-np.power(kernel_1d / sigma, 2) / 2)))
```

- The 1D kernel is then used to create a 2D kernel by taking the outer product of the 1D kernel with its transpose. The resulting 2D kernel is normalized by dividing it by the sum of its elements.

```
kernel_2d = np.outer(kernel_1d.T, kernel_1d.T)
kernel_2d *= 1.0 / np.sum(kernel_2d)
kernel = kernel_2d
```

- The **convolve\_layer()** function is applied to each color channel of the image using the generated kernel. The resulting convolved layers are stacked together using **np.dstack** to form the final blurred image.

```
# convolution
result = np.uint8(np.dstack((convolve_layer(image[:, :, 0], kernel),
                             convolve_layer(image[:, :, 1], kernel),
                             convolve_layer(image[:, :, 2], kernel))))
```

- The blurred image is displayed using **matplotlib.pyplot.imshow** and saved to disk using **PIL.Image.fromarray**.

```
plt.figure() # create a new figure object
plt.imshow(result)
plt.title("Image After Blur")
plt.show()
```

- The function serves as an entry point to the program. It loads an image using **PIL.Image.open** and converts it to a NumPy array using **np.array**. The blur\_Img function is then called to apply the Gaussian blur to the image.

## b. Sharpen Image

- **def blurforsharp(picture2):** is used to apply a blur effect to an image using a specific kernel.
  - First converts the input image (picture2) into a NumPy array using **np.array**.
  - A kernel is defined as a 3x3 array with the values [-1, -1, -1] in the first row, [-1, 9, -1] in the second row, and [-1, -1, -1] in the third row.



This kernel is commonly used for sharpening images. An empty array called **picture\_result** is created with the same shape as the input image to store the result of the blurring operation.

```
kernel = np.array([[ -1, -1, -1], [ -1, 9, -1], [ -1, -1, -1]])
picture_result = np.zeros(picture.shape)
```

- The variable **kernelDim** is set to the size of the kernel. The sum of all values in the kernel is calculated using **np.sum(kernel)** and stored in **kernel\_sum**.

```
kernelDim = kernel.shape[0]
kernel_sum = np.sum(kernel)
```

- If the sum of the kernel is not zero, indicating that the kernel is not entirely zero, the kernel is normalized by dividing it by **kernel\_sum**. This step ensures that the kernel weights are proportional and that the resulting image is not overly bright or dark. The function then loops over each element in the kernel using nested for loops. For each element, it calculates the row and column shift values (**rowShiftValue** and **colShiftValue**) based on the position of the element in the kernel. These shift values determine how much the image is rolled in each direction for the convolution operation. The **np.roll** function is used to shift the image (picture) by the calculated shift values along the rows and columns. This creates a shifted version of the image.

```
# Normalize kernel
if kernel_sum != 0:
    kernel = kernel / kernel_sum
for i in range(kernelDim):
    for j in range(kernelDim):
        rowShiftValue = int(i - kernelDim/2)
        colShiftValue = int(j - kernelDim/2)
        shiftedArray = np.roll(picture, (rowShiftValue, colShiftValue), axis=(0, 1))
        picture_result += shiftedArray * kernel[i,j]
```

- The shifted image is multiplied element-wise with the corresponding kernel element, and the result is added to **picture\_result**. This step performs the convolution operation between the image and the kernel.
- After the convolution loop, the **picture\_result** array is cast to the **np.uint8** data type, which is the expected data type for representing image pixel values.

```
picture_result = picture_result.astype(np.uint8)
```

- The pixel values in **picture2\_result** are clipped between 0 and 255 using **np.clip** to ensure they remain within the valid range for image intensity values.

```
# Clip picture2 values between 0 and 255
picture_result = np.clip(picture_result, 0, 255)
```

- **def Sharpen(picture2, name\_img):** apply the sharpening operation, display the resulting image, and save it to disk.
  - Firstly, image is converted to a NumPy array, and its shape is stored in the variable temp. A kernel is defined as a 3x3 array with all elements equal to 1, and then divided by 9 to normalize the kernel.

```
kernel = np.array([[1,1,1], [1,1,1], [1,1,1]]) / 9
```

- The **blurforsharp()** is called with the input image as an argument. This function applies a blur effect to the image using a specific kernel. The result of the blur operation is stored back in picture.
- A new array called picture\_result is created with the same shape as picture2 to store the result of the sharpening operation. The function then loops over each element in the kernel using nested for loops. For each element, it calculates the row and column shift values (rowShiftValue and colShiftValue) based on the position of the element in the kernel. The np.roll function is used to shift the blurred image (picture2) by the calculated shift values along the rows and columns. This creates a shifted version of the image. The shifted image is multiplied element-wise with the corresponding kernel element, and the result is added to picture2\_result. This step performs the convolution operation between the blurred image and the kernel.

```
# Subtract the blurred picture2 from the original picture2 to increase sharpness
picture2_result = np.zeros(picture2.shape)
for i in range(kernelDim):
    for j in range(kernelDim):
        rowShiftValue = int(i - kernelDim/2)
        colShiftValue = int(j - kernelDim/2)
        shiftedArray = np.roll(picture2, (rowShiftValue, colShiftValue), axis=(0, 1))
        picture2_result += shiftedArray * kernel[i,j]
```

- After the convolution loop, picture2\_result is reshaped to match the original image shape stored in temp. It is then cast to the np.uint8 data type.

```
picture2_result = picture2_result.reshape(temp)
picture2_result = picture2_result.astype(np.uint8)
```

- The sharpened image is displayed using **matplotlib.pyplot.imshow**. The sharpened image is saved to disk with the modified filename, using **PIL.Image.fromarray**.

```
plt.figure() # create a new figure object
plt.imshow(picture2_result)
plt.title("Image After Sharpen")
plt.show()
Image.fromarray(picture2_result).save(name_img.split('.')[0] + '_sharpen' + '.png')
```

## 6. Crop Image to Size (Crop in Center)

- *def crop\_img\_center(image, name\_img):* is used to crop image in the center of the image

- The Image.open function is used to open the image file specified by name\_img and assign it to the variable image1.
- The width and height of the image are obtained using the size attribute of the image1 object.

```
# Get the size of the image
width, height = image1.size
```

- The coordinates for cropping the center quarter of the image are calculated as follows:
  - left is set to one-fourth of the width.
  - top is set to one-fourth of the height.
  - right is set to three-fourths of the width.
  - bottom is set to three-fourths of the height.

```
# Calculate the coordinates of the center quarter of the image
left = width / 4
top = height / 4
right = 3 * width / 4
bottom = 3 * height / 4
```

- Then, we cast the coordinates to integers.

```
left = int(left)
upper = int(top)
right = int(right)
lower = int(bottom)
```

- In crop function, we are using array slicing to extract a specific region or "crop" from the image. **upper:lower** represents the range of rows we want to include in the crop. It specifies the vertical boundaries of the crop, where upper is the index of the top row (inclusive) and lower is the index of the bottom row (exclusive). **left:right** represents the range of columns we want to include in the crop. It specifies the horizontal boundaries of the crop, where left is the index of the leftmost column (inclusive) and right is the index of the rightmost column (exclusive). ":" in the third position implies that we want to include all channels or color components of the image. In this case, ":" indicates that we want to include all color channels (e.g., RGB or RGBA).

```
# Crop the image to the center quarter
cropped_image = image[upper:lower, left:right, :]
```

- The cropped image is displayed using **matplotlib.pyplot.imshow**. The cropped image is saved to disk with the modified filename, using **PIL.Image.fromarray**. Note that the **np.array** function is used to convert the **cropped\_image** object to a NumPy array before saving.

```
plt.figure() # create a new figure object
plt.imshow(cropped_image)
plt.title("Image After Crop in Center")
plt.show()
Image.fromarray(np.array(cropped_image)).save(str(name_img.split('.')[0]) + '_center_crop' + '.png')
```

## 7. Crop Image in a Circular Frame

### a. Algorithm

- Load the image using a suitable image processing library.
- Get the dimensions of the image (width and height).
- Determine the center coordinates of the image by dividing the width by 2 and the height by 2.
- Set the radius of the circular crop area. This can be determined based on a desired fraction of the image size or a specific pixel value.
- Create a new blank image with the same dimensions as the original image, but with a transparent background.
- Iterate over every pixel in the new blank image and check if it falls within the circular crop area. To do this, calculate the distance between the pixel coordinates and the center of the image using the distance formula:  $distance = \sqrt{(x - center\_x)^2 + (y - center\_y)^2}$ . If the distance is less than or equal to the radius, the pixel falls within the circular crop area.
- For each pixel that falls within the circular crop area, copy the corresponding pixel value from the original image to the new blank image.
- Save the new image with the circular crop.

### b. Implementation

- **def circular\_Crop(image, name\_img):** crop the image in a circular form, display the resulting circularly cropped image, and save it to disk
  - Firstly, it takes an image object (image) and a filename (name\_img) as input. The center and radius variables are checked. If they are not provided, the code sets center to the middle of the image and radius to the smallest distance between the center and the image walls.

```
# create ellipse mask
center=None
radius=None
d = min(image.shape[1],image.shape[0])
if center is None: # use the middle of the image
    center = (int(d/2), int(d/2))
if radius is None: # use the smallest distance between the center and image walls
    radius = min(center[0], center[1], image.shape[1]-center[0], image.shape[0]-center[1])
```

- The **np.ogrid()** is used to create a grid of coordinates for the image. The distance from each coordinate to the center is calculated using the Euclidean distance formula. A mask is created by checking if the

distance from each coordinate to the center is less than or equal to the specified radius.

```
Y, X = np.ogrid[:image.shape[0],:image.shape[1]]
xe = -(X-center[1]) + (Y-center[0])
ye = (X-center[1]) + (Y-center[0])
e = (xe)**2/((d)*(2**0.5)) + (ye)**2/((d/4)*(2**0.5))
```

- The original image is loaded and converted to a NumPy array. The circular mask is applied to a copy of the original image, resulting in a masked image. Pixels outside the circular mask are set to zero in the masked image. The original image is restored by setting the pixels outside the circular mask to the corresponding pixel values from the masked image.

```
mask = e <= radius
mask1 = e1 <= radius
mask2 = mask | mask1
image[~mask2] = 0
```

- The resulting circularly cropped image is displayed using **matplotlib.pyplot.imshow**. The circularly cropped image is saved to disk with the modified filename, using **PIL.Image.fromarray**.

```
plt.figure() # create a new figure object
plt.imshow(image)
plt.title("Image After Adjust Contrast")
plt.show()
Image.fromarray(image).save(name_img.split('.')[0] + '_ellipse_crop' + '.png')
```

## 8. Advanced: Crop Image in frame is 2 diagonal ellipses

### a. Algorithm

This algorithm creates a new image with a transparent background and copies the pixels from the original image that fall within either of the two diagonal ellipses. All other pixels outside the ellipses remain transparent. The resulting image will have a frame consisting of two diagonal ellipses.

- Load the image using a suitable image processing library.
- Get the dimensions of the image (width and height).
- Define the parameters for the two diagonal ellipses:
  - Center coordinates of each ellipse.
  - Major and minor axis lengths of each ellipse.
  - Rotation angle of each ellipse (if needed).
- Create a new blank image with the same dimensions as the original image.
- Iterate over every pixel in the new blank image and check if it falls within either of the two ellipses. To do this, calculate the relative position of the pixel with respect to each ellipse's center and use the ellipse equation to determine if the pixel is inside the ellipse. The equation for an ellipse centered at (h, k) with major axis length 'a', minor axis length 'b', and rotation

angle 'theta' is:  $((x-h)*\cos(\theta) + (y-k)*\sin(\theta))^2/a^2 + ((x-h)*\sin(\theta) - (y-k)*\cos(\theta))^2/b^2 \leq 1$ .

- For each pixel that falls within either ellipse, copy the corresponding pixel value from the original image to the new blank image.
- Save the new image with the cropped frame.

## b. Implementation

- **def ellipse\_Crop(img, name\_img):**

- Firstly, the center and radius variables are checked. If they are not provided, the code sets center to the middle of the image and radius to the smallest distance between the center and the image walls.

```
# create ellipse mask
center=None
radius=None
d = min(image.shape[1],image.shape[0])
if center is None: # use the middle of the image
    center = (int(d/2), int(d/2))
if radius is None: # use the smallest distance between the center and image walls
    radius = min(center[0], center[1], image.shape[1]-center[0], image.shape[0]-center[1])
```

- The variable d is calculated as the minimum dimension of the image (either width or height).

```
d = min(image.shape[1],image.shape[0])
```

- The **np.ogrid()** function is used to create a grid of coordinates for the image.
- Two sets of equations are used to define two ellipse shapes: e and e1. These equations represent the distance from each coordinate to the center of each ellipse shape.

```
xe1 = (X-center[1]) + (Y-center[0])
ye1 = -(X-center[1]) + (Y-center[0])
e1 = (xe1)**2/((d)*(2**0.5)) + (ye1)**2/((d/4)*(2**0.5))
```

- Masks are created by checking if the distance from each coordinate to the center falls within the specified radius for each ellipse. The masks are combined using **logical OR (|)** to create a single mask that includes both ellipses. Pixels outside the combined mask are set to zero in the image, effectively cropping the image to the ellipse shape.

```
mask = e <= radius
mask1 = e1 <= radius
mask2 = mask | mask1
image[~mask2] = 0
```

- The resulting cropped image is displayed using **matplotlib.pyplot.imshow**. The cropped image is saved to disk with the modified filename, using **PIL.Image.fromarray**.

## 9. Main Function

- **def processing\_option(image, name\_img, option):** The function based on the option value, it performs different image processing operations and measures the running time for each operation.





```

print('***** MAIN MENU *****')
print('#0  Do all tasks')
print('#1  Adjust Brightness')
print('#2  Adjust Contrast')
print('#3  Flip Image (Vertical/Horizontal)')
print('#4  Convert from RGB to Grayscale/Sepia')
print('#5  Blur and Sharpen Image')
print('#6  Crop Image to Size (Crop in Center)')
print('#7  Crop Image in a Circular Frame')
print('#8  Advanced: Crop Image in frame is 2 diagonal ellipses')
print('*****')




option= int(input('Enter option : '))
processing_option(image, name_img, option)




```



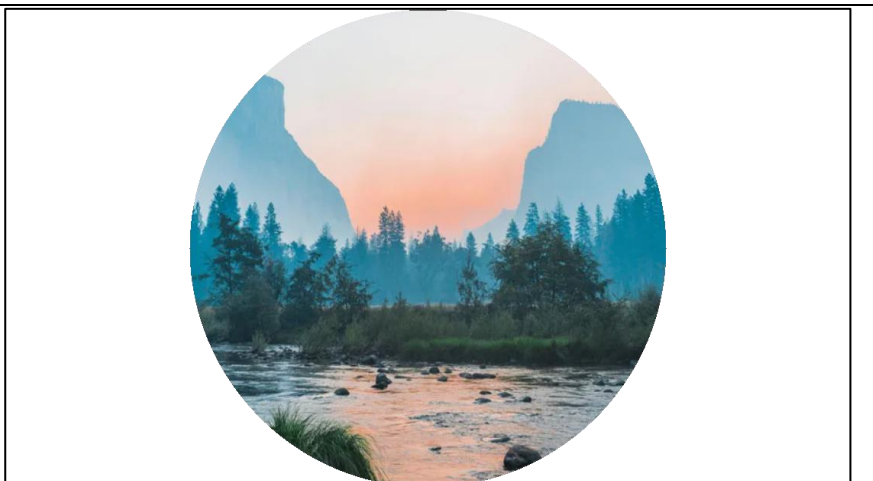
## V. RESULT

Function	Image
<b>Original image</b>	
<b>Adjust Brightness</b>	



<b>Adjust Contrast</b>	
<b>Flip Vertical</b>	
<b>Flip Horizontal</b>	

<p><b>Convert to Grayscale</b></p>	
<p><b>Convert to Sepia</b></p>	
<p><b>Blur Image</b></p>	

<p><b>Sharpen Image</b></p>	
<p><b>Crop in center</b></p>	
<p><b>Circular Frame</b></p>	





## VI. REFERENCE

1. **Image Processing with Python — Blurring and Sharpening for Beginners**  
<https://towardsdatascience.com/image-processing-with-python-blurring-and-sharpening-for-beginners-3bcebec0583a>
2. **Processing an image to sepia tone in Python**  
<https://stackoverflow.com/questions/36434905/processing-an-image-to-sepia-tone-in-python>
3. **Python PIL | Image.crop() method**  
<https://www.geeksforgeeks.org/python-pil-image-crop-method/>
4. **Numpy Convolve**  
<https://numpy.org/doc/stable/reference/generated/numpy.convolve.html>