

**VIETNAM NATIONAL UNIVERSITY HCMC  
UNIVERSITY OF SCIENCE  
FACULTY OF INFORMATION TECHNOLOGY**

—o0o—



---

## **Project 2: Image Processing**

---

**Subject: Applied Mathematics and Statistics**

*Lecturer:*

**Mr. Vũ Quốc Hoàng  
Mr. Nguyễn Văn Quang Huy  
Mr. Nguyễn Ngọc Toàn  
Ms. Phan Thị Phương Uyên**

*Class:*

**22CLC05**

*Student:*

**22127119 – Hồ Phước Hoàn**

**HCMC, 7/2024**

# Table of Contents

- I. Tổng quan .....3
- II. Thực hiện .....3
  - 1. Cài đặt chi tiết.....3
  - 2. Các hàm chức năng .....3
- 3. Bảng đánh giá và hình ảnh ..... 15
- Tài liệu tham khảo ..... 18

## I. Tổng quan

- [1] Trong khoa học máy tính, xử lý ảnh là việc sử dụng các thuật toán trên máy tính để thực hiện xử lý hình ảnh trên hình ảnh kỹ thuật số. Là một danh mục con hoặc lĩnh vực xử lý tín hiệu số, xử lý hình ảnh kỹ thuật số có nhiều lợi thế so với xử lý hình ảnh tương tự. Nó cho phép phạm vi thuật toán áp dụng rộng hơn nhiều, được áp dụng cho dữ liệu đầu vào và có thể tránh được các vấn đề như sự tích tụ nhiễu và méo tín hiệu trong quá trình xử lý. Vì hình ảnh được xác định theo hai chiều (có thể nhiều hơn) nên việc xử lý hình ảnh kỹ thuật số có thể được mô hình hóa dưới dạng các hệ thống đa chiều. Việc tạo ra và phát triển xử lý ảnh số chủ yếu chịu ảnh hưởng của ba yếu tố: thứ nhất, sự phát triển của máy tính; thứ hai, sự phát triển của toán học (đặc biệt là việc tạo ra và cải thiện lý thuyết toán học rời rạc); thứ ba, nhu cầu về ứng dụng rộng rãi trong môi trường, nông nghiệp, quân sự, công nghiệp và khoa học y tế đã tăng lên.
- Trong đồ án này, em thực hiện một số thuật toán xử lý ảnh căn bản như chỉnh sáng, chỉnh độ tương phản, làm mờ, ...

## II. Thực hiện

### 1. Cài đặt chi tiết

Source code được viết bằng ngôn ngữ Python, thực hiện trên môi trường Jupyter Notebook. Trong Source code có sử dụng các thư viện cần thiết bên ngoài như:

- Numpy: thư viện dùng để tính toán trong thuật toán
- Matplotlib: thư viện dùng để hiển thị ảnh và so sánh hình ảnh gốc và ảnh đã nén màu
- PIL: thư viện dùng để mở ảnh và lưu ảnh

### 2. Các hàm chức năng

#### a) *read\_img*

- Tham số đầu vào:
  - + 'img\_path': đường dẫn file đầu vào (str)
- Kết quả trả về: data của ảnh vừa đọc có kiểu dữ liệu Image.Image trong PIL
- Mô tả:

Hàm đọc ảnh từ đường dẫn file ảnh bên ngoài.

Hàm cần lưu ý:

+ *Image.open(img\_path)*: hàm đọc ảnh từ đường dẫn file ảnh

#### b) *show\_img*

- Tham số đầu vào:
  - + 'img': ảnh 2 chiều (np.ndarray)
  - + 'title': tiêu đề ảnh (str)

- Kết quả trả về: không có

- Mô tả:

Hàm hiển thị hình ảnh, có hiển thị tiêu đề ảnh.

Hàm cần lưu ý:

- + `plt.imshow(img)`: Hàm chọn ảnh để show
- + `plt.title(title)`: Hàm ghi tiêu đề cần show trên ảnh
- + `plt.show()`: Hàm thực hiện show ảnh

c) *save\_img*

- Tham số đầu vào:

- + 'img': ảnh 2 chiều (np.ndarray)
- + 'img\_path': đường dẫn file ảnh muốn lưu (str)

- Kết quả trả về: không có

- Mô tả:

Hàm lưu ảnh theo đường dẫn file đưa vào xuống máy tính.

Hàm cần lưu ý:

- + `Image.fromarray(img)`: Hàm chuyển đổi kiểu dữ liệu mảng (np.ndarray) sang kiểu dữ liệu Image.Image của PIL

d) *adjustBrightness*

- Tham số đầu vào:

- + 'img': ảnh 2 chiều (np.ndarray)
- + 'alpha': độ tăng/giảm sáng cần điều chỉnh ảnh (float). Mặc định là 50.0

- Kết quả trả về: ảnh đã tăng/giảm sáng (np.ndarray)

- Mô tả:

Hàm tăng sáng/giảm sáng cho ảnh theo mức tăng giảm alpha cho trước.

Đầu tiên, chuyển đổi các phần tử trong mảng từ kiểu dữ liệu np.uint8 sang np.float32 để tránh bị tràn số khi cộng với giá trị alpha. Sau đó giới hạn lại giá trị trong đoạn [0, 255] cho từng kênh màu của mỗi pixel. Cuối cùng chuyển kiểu dữ liệu phần tử sang np.uint8 để ảnh có thể hiển thị được. Càng tăng giá trị alpha ảnh càng sáng, và ngược lại.

Hàm cần lưu ý:

- + `img.astype(np.float32)`: Hàm chuyển đổi kiểu dữ liệu sang np.float32
- + `np.clip(img + alpha, 0, 255)`: Hàm giới hạn giá trị của các phần tử trong đoạn [0, 255]

e) *adjustContrast*

- Tham số đầu vào:

- + 'img': ảnh 2 chiều (np.ndarray)

- + ‘alpha’: độ tăng/giảm tương phản cần điều chỉnh ảnh (float). Mặc định là 2.0
- Kết quả trả về: ảnh đã tăng/giảm tương phản (np.ndarray)
- Mô tả:  
Đầu tiên, chuyển đổi các phần tử trong mảng từ kiểu dữ liệu np.uint8 sang np.float32 để tránh bị tràn số khi nhân với giá trị alpha. Sau đó giới hạn lại giá trị trong đoạn [0, 255] cho từng kênh màu của mỗi pixel. Cuối cùng chuyển kiểu dữ liệu phần tử sang np.uint8 để ảnh có thể hiển thị được. Càng tăng giá trị alpha ảnh càng tương phản cao, và ngược lại.

f) *flipImage*

- Tham số đầu vào:
  - + ‘img’: ảnh 2 chiều (np.ndarray)
  - + ‘axis’: trục đối xứng (bool)
    - 0: Lật (lên/xuống) theo chiều dọc (đối xứng ngang). Mặc định là 0
    - 1: Lật (trái/phải) theo chiều ngang (đối xứng dọc)
- Kết quả trả về: ảnh đã lật (np.ndarray)
- Mô tả:  
Lật ảnh theo chiều mong muốn.

Hàm cần lưu ý:

- + *np.flip(img, axis)*: hàm lật ảnh theo chiều axis

g) *rgb\_gray*

- Tham số đầu vào:
  - + ‘img’: ảnh 2 chiều (np.ndarray)
- Kết quả trả về: ảnh đã đổi sang màu xám (np.ndarray)
- Mô tả:  
Đổi màu ảnh từ màu rgb sang màu xám dùng kỹ thuật Luminosity để mang lại màu sắc chuẩn hơn những kỹ thuật khác.  
Đầu tiên, chuyển đổi các phần tử trong mảng từ kiểu dữ liệu np.uint8 sang np.float32 để tránh bị tràn số khi nhân rồi cộng các giá trị kênh màu. Sau đó ta tính tích vô hướng cho mảng N-D array và 1-D array. Mảng N-D array là mảng 3 chiều (n, m, 3) của ảnh, mảng 1-D array là mảng 1 chiều với các giá trị [299/1000, 587/1000, 114/1000] trong kỹ thuật Luminosity và tính theo công thức [5]:  $0.299 * R + 0.587 * G + 0.114 * B$ . Trong np.dot, vì ta nhận thấy khi áp dụng với N-D array và 1-D array, nó sẽ tính tích vô hướng với trục cuối cùng của N-D array và 1-D array, do đó nó sẽ đúng với công thức ta muốn thực hiện. Tuy nhiên sau khi tính tích vô hướng xong thì mảng chứa ảnh sẽ thành 2 chiều (n, m) vì tính tích vô hướng nó sẽ cộng lại các giá trị sau khi nhân, nên ta sẽ tạo ra một chiều mới cho các kênh màu bằng [...,

`np.newaxis]`, thì shape sẽ là  $(n, m, 1)$  và ta nhận thấy ảnh xám thì các kênh màu trên 1 pixel thì giống nhau nên sẽ dùng hàm `repeat(3, axis=2)` để tạo ra 3 giá trị giống hệt giá trị màu vừa tính xong, khi đó shape sẽ là  $(n, m, 3)$  đúng với hình dạng ban đầu của ảnh, `axis=2` để chỉ thực hiện hàm này trên chiều cuối cùng, là chiều chứa các kênh màu. Cuối cùng chuyển kiểu dữ liệu phần tử sang `np.uint8` để ảnh có thể hiển thị được. Thực ra với ảnh xám  $(n, m)$  thì vẫn có thể hiển thị ảnh bình thường, tuy nhiên, để nhất quán thì ta sẽ chuyển lại từ  $(n, m)$  sang  $(n, m, 3)$ .

#### Hàm cần lưu ý:

- + `np.dot(img, [0.299, 0.587, 0.114])[..., np.newaxis]`: tính tích vô hướng ở chiều cuối cùng rồi reshape ảnh sang shape  $(n, m, 1)$ .
- + `repeat(3, axis=2).astype(np.uint8)`: tạo ra 3 giá trị kênh màu y hệt giá trị vừa tính toán ở chiều cuối cùng rồi chuyển đổi kiểu dữ liệu sang `np.uint8`.

#### *h) rgb\_sepia*

- Tham số đầu vào:
  - + 'img': ảnh 2 chiều (`np.ndarray`)
- Kết quả trả về: ảnh đã đổi sang màu sepia (`np.ndarray`)
- Mô tả:

Đổi màu ảnh từ màu rgb sang màu sepia

Đầu tiên, chuyển đổi các phần tử trong mảng từ kiểu dữ liệu `np.uint8` sang `np.float32` để tránh bị tràn số khi nhân rồi cộng các giá trị kênh màu. Với mảng màu để dùng thuật toán chuyển đổi màu sepia sẽ là mảng 2 chiều:

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

Ta áp dụng công thức sau [7]:

$$tr = 0.393R + 0.769G + 0.189B$$

$$tg = 0.349R + 0.686G + 0.168B$$

$$tb = 0.272R + 0.534G + 0.131B$$

Mỗi mảng 1 chiều của sepia sẽ để tính một kênh màu cho mỗi pixel nên ta sẽ nghĩ đến việc chuyển đổi shape của ảnh sang ảnh 1 chiều  $(n*m, 3)$  rồi nhân với ma trận chuyển vị của mảng màu sepia. Lý giải vì sao lại reshape rồi nhân với ma trận chuyển vị của sepia array, là vì sepia array là mảng 2 chiều còn ảnh là mảng 3 chiều. Ta làm việc trên từng pixel của ảnh để nhân với từng mảng 1 chiều của sepia và vì vậy để thực hiện nhân 2 ma trận ta cần chuyển vị ma trận sepia để đáp ứng đúng công thức. Khi chuyển đổi sang  $(n*m, 3)$  ảnh sẽ thành 1 ma trận với  $n*m$  dòng và mỗi dòng có 3 cột tức là các kênh màu của từng pixel. Khi nhân với ma trận

chuyển vị sepia, nó sẽ là nhân 2 ma trận, và với nhân 2 ma trận thì dòng của ma trận này sẽ nhân cột của ma trận kia và trả về 3 giá trị kênh màu cho từng dòng sau khi tính. Tức là mỗi pixel trên 1 dòng và đúng với mong muốn của chúng ta, shape sau khi tính vẫn sẽ giữ  $(n*m, 3) * (3, 3) \rightarrow (n*m, 3)$ . Ta chỉ cần giới hạn lại giá trị sau khi tính trong đoạn  $[0, 255]$  và chuyển đổi từng phần tử sang kiểu np.uint8 rồi reshape lại ma trận theo shape ban đầu của ảnh là kết thúc.

#### Hàm cần lưu ý:

- + `img.reshape(-1, 3)`: chuyển đổi shape của ma trận sang  $(n * m, 3)$
- + `img_1d @ sepia_arr.T`: ma trận ảnh 1 chiều nhân với ma trận chuyển vị sepia

#### i) `gaussian_kernel`

- Tham số đầu vào:
  - + 'size': kích thước của kernel, nên là số lẻ (int). Mặc định là 5.
  - + 'sigma': độ lệch chuẩn trong phân phối gaussian (phân phối chuẩn) (float). Mặc định là: 1.0
- Kết quả trả về: mảng 1 chiều các giá trị đã tính hàm Gaussian kernel (np.ndarray)
- Mô tả:

Hàm tạo ra mảng Gaussian kernel 1 chiều với kích thước cụ thể và độ lệch chuẩn cụ thể.

Vì ta sẽ convolve (tích chập) trên 1 chiều nên chỉ dùng mảng 1 chiều để tính toán các giá trị trên 1 trục. Có thể là trục x hoặc trục y. Để lí giải tại sao lại convolve trên 1 chiều mà không phải 2: đầu tiên: ta dùng np.convolve trong numpy để convolve mà không cần viết hàm thủ công, vừa nhanh vừa gọn. Thứ 2: trong gaussian blur convolution thì convolve trên từng chiều sẽ cho ra tốc độ nhanh hơn và vẫn giữ nguyên được kết quả như convolve 2 chiều.

Ta áp dụng hàm kernel sau để tính: [15]

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

Đầu tiên ta tạo ra mảng các phần tử cách đều nhau đi từ giá trị  $-(size-1)/2$  tới  $(size-1)/2$  với size các giá trị. Sau đó dùng hàm kernel để tính cho từng giá trị trong đó.

#### Hàm cần lưu ý:

- + `np.linspace(-(size - 1) / 2., (size - 1) / 2., size)`: hàm tạo ra các phần tử cách đều nhau đi từ giá trị  $-(size-1)/2$  tới  $(size-1)/2$  với size các giá trị
- + `np.exp()`: hàm  $e^{\dots}$
- + `np.sqrt()`: hàm  $\sqrt{\dots}$

- + `np.pi`: giá trị  $\pi$
- j) `convolve`
  - Tham số đầu vào:
    - + ‘img’: ảnh 2 chiều (`np.ndarray`)
    - + ‘kernel’: mảng 1 chiều các giá trị đã tính hàm kernel xong (`np.ndarray`)
  - Kết quả trả về: ảnh đã được convolve  $\rightarrow$  làm mờ (vì dùng gaussian kernel để làm mờ) (`np.ndarray`)
  - Mô tả:
 

Convolve ảnh với hàm kernel cho trước.

Hàm convolve của image processing [8], đây là dạng discrete convolution nhưng ở 2 chiều

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

- +  $g(x, y)$ : ảnh sau khi chỉnh
- +  $\omega$ : mảng 1 chiều chứa các giá trị đã tính kernel
- +  $f(x, y)$ : ảnh 2 chiều ban đầu.

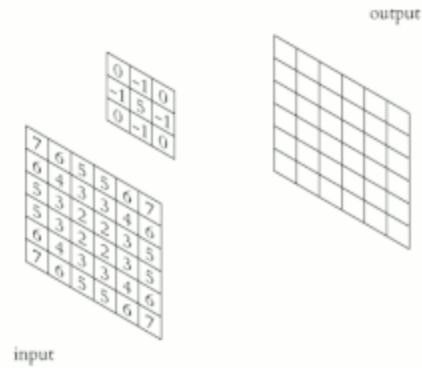
Vì ở đây ta dùng hàm kernel trên 1 chiều và hàm convolve trên 1 chiều nên discrete convolve trên 1 chiều sẽ là [17]:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n - m],$$

Tương ứng với hàm `np.convolve` [14]:

$$(a * v)_n = \sum_{m=-\infty}^{\infty} a_m v_{n-m}$$





### Convolve 2 chiều [8]

Việc dùng `np.convolve(a, b, mode='same')` trên 1 chiều sẽ hoạt động như sau. Đầu tiên nó sẽ lấy ra độ dài lớn nhất giữa 2 mảng a, b, và mảng xuất ra sẽ có độ dài đó. Tiếp theo sẽ lật mảng nhỏ hơn, với kích thước mảng a, đặt là M, kích thước mảng b, đặt là N thì sẽ có M+N-1 convolution và sẽ convolve theo từ trái qua phải theo cửa sổ trượt.

```
u = [1, 2, 3, 2, 1]; v = [1, 0, -1];
```

```
conv(u, v) =      1      2      2      0     -2     -2     -1
```

```
conv(u, v, 'same') =      2      2      0     -2     -2
```

### Discrete Convolution 1 chiều trên 2 mảng

Và ta sẽ convolve lần lượt trên từng dòng rồi đến từng cột. Ngoài ra để ta mong muốn số phần tử không tăng thêm cũng không mất đi nên sẽ dùng `mode='same'` trong `np.convolve`.

Ta sẽ tạo ra một mảng tạm để lưu các giá trị sau khi convolve dòng xong rồi sẽ convolve trên cột dùng mảng tạm đấy. Cuối cùng sẽ ra kết quả sau khi đã convolve lần lượt từng chiều với kết quả tương tự như convolve 2 chiều như hình trên.

#### Hàm cần lưu ý:

- + `np.zeros_like(img, dtype=np.float32)`: tạo ra 1 mảng toàn 0 với shape = shape của ảnh và mang kiểu dữ liệu `np.float32` để tính toán không bị tràn số
- + `np.convolve(img[i, :, j], kernel, mode='same')`: convolve trên từng dòng và trên từng kênh màu với mảng 1 chiều kernel. Hàm này không thay đổi số phần tử trong mảng trả ra mà lấy `max(m, n)`: số phần tử lớn nhất trong 2 mảng.

+ `np.convolve(temp_result[:, i, j], kernel, mode='same')`: convolve trên từng cột và dùng kết quả của convolve trên dòng để convolve trên cột

k) *unsharp\_mask*

- Tham số đầu vào:
  - + 'img': ảnh 2 chiều (np.ndarray)
  - + 'size': kích thước của kernel, nên là số lẻ (int). Mặc định là 5.
  - + 'sigma': độ lệch chuẩn trong phân phối gaussian (phân phối chuẩn) (float). Mặc định là: 1.0
  - + 'strength': độ mạnh trong làm nét ảnh (float), càng tăng ảnh càng nét. Mặc định là 1.5
- Kết quả trả về: ảnh đã làm nét (np.ndarray)
- Mô tả:

Hàm làm nét ảnh sử dụng kỹ thuật unsharp masking.

Công thức unsharp masking [18]:

$$\text{sharpened} = \text{original} + (\text{original} - \text{blurred}) \times \text{amount}.$$

+ Sharpened: ảnh đã làm nét

+ original: ảnh ban đầu

+ blurred: ảnh đã làm mờ

+ amount: độ mạnh trong làm nét ảnh

Ban đầu ta sẽ tạo ra ảnh mờ dùng kỹ thuật gaussian blur, sau đó áp dụng công thức unsharp masking để làm mờ rồi lại giới hạn giá trị của kênh màu và chuyển đổi sang kiểu dữ liệu np.uint8. Vì những biến img và blurred\_img cùng shape nên việc tính toán cực kì dễ dàng.

l) *crop\_img\_center*

- Tham số đầu vào:
  - + 'img': ảnh 2 chiều (np.ndarray)
  - + 'height': chiều cao ảnh cần cắt (float). Mặc định là: 256.0
  - + 'width': chiều rộng ảnh cần cắt (float). Mặc định là: 256.0

Kết quả trả về: ảnh đã cắt ở phần trung tâm (np.ndarray)

- Mô tả:

Cắt ảnh ở phần trung tâm của ảnh ban đầu với chiều dài, rộng cho trước.

Đầu tiên ta ràng buộc giá trị chiều dài và rộng cho trước phải  $> 0$  và  $\leq$  chiều dài, rộng ban đầu của hình ảnh gốc. Sau đó ta tính số đo của ảnh cắt:

+ top:  $(\text{chiều dài gốc} - \text{chiều dài cho trước})/2$

+ bottom:  $(\text{chiều dài gốc} + \text{chiều dài cho trước})/2$

+ left:  $(\text{chiều rộng gốc} - \text{chiều rộng cho trước})/2$

+ right:  $(\text{chiều rộng gốc} + \text{chiều rộng cho trước})/2$

Sau đó trả về mảng các hình ảnh với số dòng chạy từ top - bottom và số cột chạy từ left – right  
“img[top:bottom, left:right]”.

m) *crop\_img\_circular\_frame*

- Tham số đầu vào:

+ ‘img’: ảnh 2 chiều (np.ndarray)

Kết quả trả về: ảnh đã cắt khung tròn ở giữa (np.ndarray)

- Mô tả:

Cắt khung tròn ở giữa ảnh, phần bên ngoài khung tròn sẽ có màu đen, phần trong là nội dung ảnh

Ở đây, ý tưởng là sẽ tạo một mảng có kích thước giống ảnh nhưng chứa các giá trị true, false (mask) rồi nhân lớp mask đó với mảng chứa ảnh và trả về ảnh đã cắt hình tròn ở giữa. Nếu pixel nào mang giá trị true thì sẽ hiển thị pixel đó, false thì pixel mang giá trị (0, 0, 0). Đầu tiên ta tạo ra trục tọa độ ax, ay ứng với độ dài của width và height của ảnh gốc. Sau đó ta tạo ra tâm đường tròn dựa trên tâm của ảnh, và bán kính của đường tròn dựa vào min(dài/2, rộng/2). Sau đó ta tạo ra 1 lớp mask dựa theo công thức đường tròn:

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

(a, b): tâm đường tròn

c: bán kính đường tròn

Nếu phần tử nào trong ảnh thỏa công thức trên thì trong mask sẽ có giá trị true, ngược lại là false. Ngoài ra vì ax là trục x với shape (1, m) và ay là trục y với shape (n, 1) nên khi thực hiện  $(x - a)^2 + (y - b)^2$  sẽ tạo ra lớp mask với shape (n, m) đúng với kích thước ảnh 2 chiều ban đầu.

Tiếp theo ta sẽ tạo ra một mảng kết quả mang các giá trị 0 và có shape giống với shape của ảnh ban đầu để làm ảnh kết quả. Và ta thực hiện vòng for trên 3 kênh màu của pixel để gán img[:, :, c] \* mask vào cho từng kênh màu của mỗi pixel với c là từng kênh màu r, g, b. img[:, :, c] trả về mảng 2 chiều cùng shape với mask nên broadcasting được.

Hàm cần lưu ý:

+ *np.ogrid[:height, :width]*: tạo ra trục x với giá trị chạy từ 0 – (width – 1) và shape = (1, width), trục y với giá trị chạy từ 0 – (height – 1) và shape = (height, 1).

n) *crop\_img\_2\_crossed\_ellipses\_frame*

- Tham số đầu vào:

+ ‘img’: ảnh 2 chiều (np.ndarray)

+ ‘angle’: số đo góc của elip (float). Mặc định là:  $\frac{\pi}{3}$

Kết quả trả về: ảnh đã cắt 2 khung elip đối xứng nhau qua trục y (np.ndarray)

- Mô tả:

Cắt khung 2 elip đối xứng nhau qua trục y ở giữa ảnh, phần bên ngoài khung sẽ có màu đen, phần trong là nội dung ảnh

Ở đây, ý tưởng tương tự như cắt khung tròn chỉ khác là công thức xác định elip là: [20]

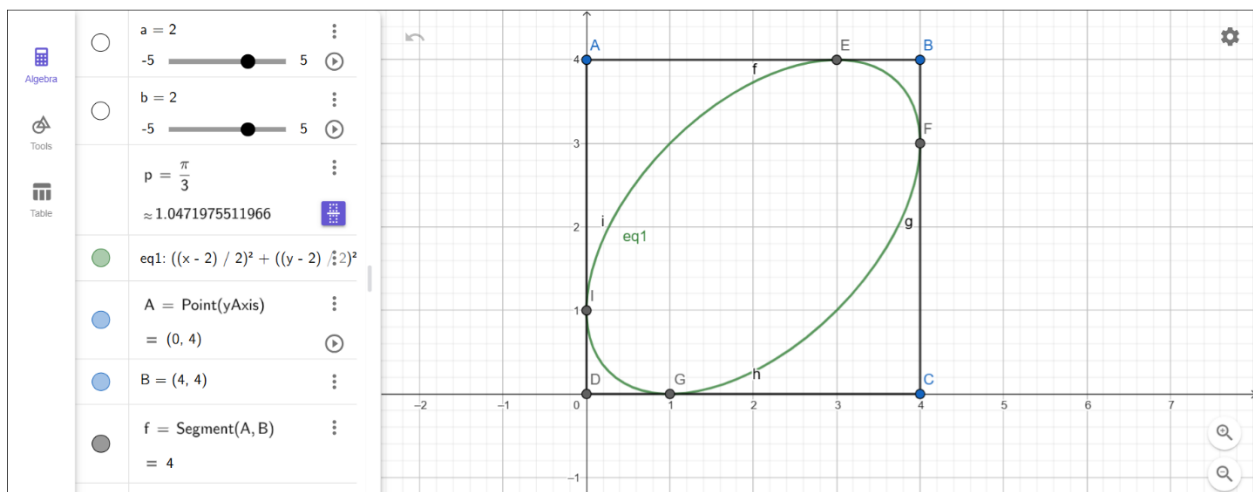
$$\left(\frac{x-x_0}{a}\right)^2 + \left(\frac{y-y_0}{b}\right)^2 - \frac{2(x-x_0)(y-y_0)}{ab} \cos \varphi \leq \sin^2 \varphi$$

$(x_0, y_0)$ : tâm hình elip

$a$  = chiều rộng ảnh / 2

$b$  = chiều dài ảnh / 2

$\varphi$ : góc quay của elip (angle)



Với hàm elip xiên này thì giá trị của  $a$  và  $b$  thay đổi theo ảnh gốc và  $a = b$  thì elip sẽ nằm đều trong ảnh hình vuông, và góc của elip để giá trị  $\frac{\pi}{3}$  thì sẽ giao với hình vuông tại 4 điểm với số đo đẹp. Có thể thay đổi số đo góc bất kì, với  $\frac{\pi}{2}$  elip sẽ thành hình tròn nếu cả  $a = b$ .

Điểm cần lưu ý ở đây chỉ là sau khi áp dụng công thức trên trong hàm tính toán thì để 2 elip đối xứng ta cần `mask += np.fliplr(mask)`, tức là cộng với phần lật ngang của mảng để mask có các phần tử đối xứng qua trục dọc.

Hàm cần lưu ý:

+ `np.fliplr()`: lật mảng theo trục dọc

o) `bicubic_kernel`

- Tham số đầu vào:

+ 'x': giá trị đầu vào hoặc mảng đầu vào cần tính kernel (`np.ndarray` hoặc `float`)

Kết quả trả về: giá trị hoặc mảng với các giá trị đã tính kernel (`np.ndarray` hoặc `float`)

- Mô tả:

Hàm tính giá trị dùng bicubic kernel được dùng trong bicubic interpolation (Nội suy bicubic)

Hàm bicubic kernel [22]:

$$W(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1 & \text{for } |x| \leq 1, \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & \text{for } 1 < |x| < 2, \\ 0 & \text{otherwise,} \end{cases}$$

Ta sử dụng np.where thay if, else để mang lại hiệu quả tốc độ tốt hơn khi sử dụng mảng hoặc giá trị của numpy. Trong np.where(a, b, c), nếu a là điều kiện đúng thì thực hiện b và ngược lại thực hiện c. Tận dụng điều này ta dùng với công thức tính kernel bên trên. Để ý thấy giá trị  $a = -0.5$ , đó là tham số điều chỉnh của bicubic kernel. Tham số này quyết định độ mượt của quá trình nội suy. Trong đa số trường hợp và để thuật toán chạy tốt thì người ta đặt  $a = -0.5$  thay vì  $a = -0.75$

Hàm cần lưu ý:

+ np.where(a, b, c): nếu a đúng thì thực hiện b, ngược lại thực hiện c

p) *bicubic\_interpolation*

- Tham số đầu vào:

+ 'img': ảnh 2 chiều (np.ndarray)

+ 'zoom\_factor': hệ số phóng to/thu nhỏ ảnh (float). Mặc định là 2.0

Kết quả trả về: ảnh đã zoom với shape = (int(height \* zoom\_factor), int(width \* zoom\_factor), channels)

- Mô tả:

Nội suy bicubic trên ảnh. Mang lại hiệu ứng tốt hơn và ảnh trông mượt hơn đa số các thuật toán khác.

Đầu tiên ta sẽ tạo ra mảng với shape = (int(height \* zoom\_factor), int(width \* zoom\_factor), channels) để chứa ảnh zoom. Tiếp theo ta tạo ra lưới tọa độ các trục x, y của ảnh zoom sau đó ta chuyển các tọa độ này về tọa độ tương ứng trên ảnh gốc bằng cách chia cho zoom\_factor.

Sau đó ta lấy giá trị sàn của mỗi điểm ảnh, chuyển sang tọa độ số nguyên và giới hạn giá trị để nó không vượt qua ngoài biên của ảnh gốc.  $x_0, y_0$  đại diện cho các vị trí của điểm ảnh gốc gần nhất trong lưới 4x4 điểm ảnh lân cận. Điều này là cần thiết cho quá trình nội suy vì chúng ta sẽ sử dụng các điểm ảnh gần nhất này để tính giá trị của điểm ảnh mới. Sau đó thực hiện  $d_x, d_y = a_x - x_0, a_y - y_0$  để tính khoảng cách từ các tọa độ trên ảnh zoom đến các điểm ảnh gần nhất  $(x_0, y_0)$  trên ảnh gốc. Trong nội suy bicubic, trọng số của các điểm ảnh lân cận phụ thuộc vào khoảng cách từ điểm cần nội suy đến các điểm ảnh lân cận.  $d_x, d_y$  cung cấp thông tin khoảng cách này, giúp tính toán trọng số dựa trên hàm kernel bicubic.

Ta lưu mảng 'weights' để chứa trọng số cho từng điểm ảnh lân cận. Tiếp theo ta lặp qua tất cả các điểm ảnh lân cận trong lưới 4x4 và tính trọng số cho từng điểm ảnh lân cận bằng cách nhân hai giá trị kernel theo hai chiều dựa trên công thức nội suy bicubic 2 chiều sau. Phép nhân  $\text{bicubic\_kernel}(dx - n) * \text{bicubic\_kernel}(dy - m)$  là cách để tính trọng số 2d cho mỗi pixel trong lưới 4x4 xung quanh điểm cần nội suy. Đây là kết quả của việc áp dụng nội suy bicubic riêng biệt theo cả hai hướng x và y, sau đó kết hợp chúng lại. Nội suy bicubic 2D có thể được xem như là tích của hai nội suy 1d độc lập. n là chỉ số của điểm ảnh lân cận theo trục x, chạy từ -1 đến 2. m là chỉ số của điểm ảnh lân cận theo trục y, chạy từ -1 đến 2. dx - n và dy - m tính toán khoảng cách từ điểm cần nội suy đến mỗi pixel trong lưới 4x4.

Cuối cùng ta sẽ lặp qua từng kênh màu của ảnh. Rồi lặp qua tất cả các điểm ảnh lân cận trong lưới 4x4, tính giá trị của mỗi pixel mới dựa vào 16 pixel lân cận. `image[y, x, c]` lấy giá trị pixel từ ảnh gốc tại vị trí (y, x) và kênh màu c. 'weights[m, n]' là trọng số tương ứng với pixel này (đã tính trước đó). Nhân giá trị pixel với trọng số và cộng vào kết quả. Lý do cộng là vì chúng ta đang tính tổng có trọng số của 16 pixel lân cận. Mỗi lần lặp, chúng ta cộng dồn đóng góp của một pixel vào kết quả cuối cùng.

#### Hàm cần lưu ý:

- + `np.zeros((new_h, new_w, channels))`: tạo một mảng các số 0 với shape cho trước.
- + `np.floor()`: lấy giá trị sàn

#### q) *modify\_file\_path*

- Tham số đầu vào:

- + 'original\_path': đường dẫn file đầu vào, có tên file và đuôi file (str)
- + 'new\_suffix': đoạn chữ cần thêm vào tên file xuất, ví dụ: \_blur, \_sharp, ... (str)
- + 'save\_path': đường dẫn file xuất, là đường dẫn tới folder (str)

Kết quả trả về: đường dẫn đến nơi xuất file, có cả tên file và đuôi file cần xuất.

- Mô tả:

Hàm hỗ trợ cho việc thêm tên phụ trợ vào đuôi tên file xuất. Đầu tiên tìm '/' cuối cùng trong original\_path, tìm '.' cuối cùng trong 'original\_path'. Sau đó cộng lại lần lượt save\_path + original\_path[last\_slash\_idx:last\_dot\_idx] + new\_suffix + original\_path[last\_dot\_idx:] để lấy đường dẫn file xuất cuối cùng.

#### Hàm cần lưu ý:

- + `.rfind('/')`: tìm dấu '/' cuối cùng trong đoạn str

#### r) *main*

- Tham số đầu vào: không

Kết quả trả về: không

- Mô tả:

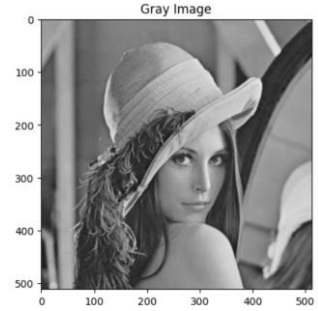
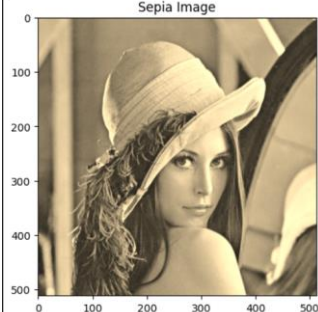
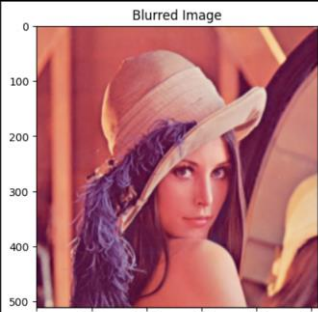

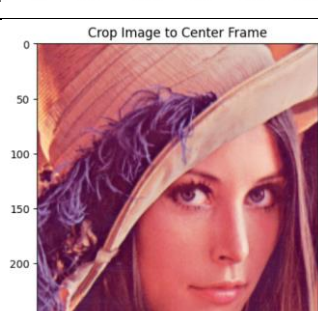
Nhận file đầu vào, đường dẫn tới folder lưu, tạo menu để lựa chọn chức năng.

Thực hiện từng chức năng cho đến khi người dùng thoát.

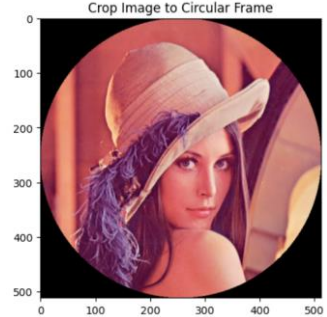

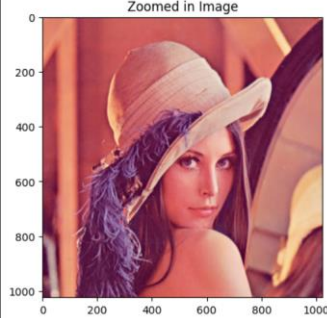
Show file và lưu file mỗi khi thực hiện 1 chức năng.

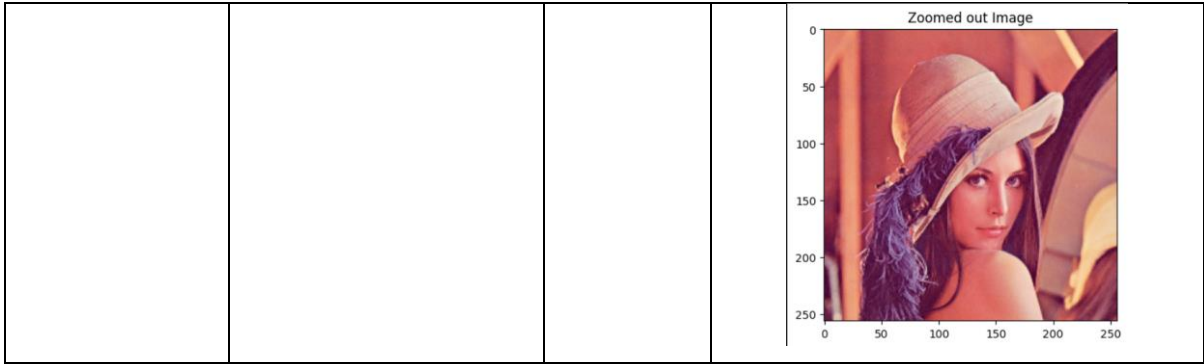
### 3. Bảng đánh giá và hình ảnh

STT	Chức năng/Hàm	Mức độ hoàn thành	Ảnh kết quả
1	Thay đổi độ sáng	100%	
2	Thay đổi độ tương phản	100%	
3.1	Lật ảnh ngang	100%	
3.2	Lật ảnh dọc	100%	

4.1	RGB thành ảnh xám	100%	
4.2	RGB thành ảnh sepia	100%	
5.1	Làm mờ ảnh	100%	
5.1	Làm sắc nét ảnh	100%	
6	Cắt ảnh theo kích thước	100%	



7.1	Cắt ảnh theo khung tròn	100%	
7.2	Cắt ảnh theo khung elip	100%	
8	Hàm main	100%	<pre> Image Processing Menu: 0. Execute all function 1. Adjust brightness 2. Adjust contrast 3. Flip (vertically / horizontally) 4. Rgb - (gray / sepia) 5. Blur / sharpen 6. Crop to size (from the center) 7. Crop to (circular / 2 crossed eclipses) frame 8. Zoom in / zoom out 9. Exit </pre>
9	Phóng to/Thu nhỏ 2x	100%	<p>Phóng to</p>  <p>Thu nhỏ</p>



## Tài liệu tham khảo

- [1] wikipedia, "Digital image processing," [Online]. Available: [https://en.wikipedia.org/wiki/Digital\\_image\\_processing](https://en.wikipedia.org/wiki/Digital_image_processing). [Accessed 31 7 2024].
- [2] "Science meets Photography: Enhancing Image Brightness with Python and NumPy - A Deep Dive," [Online]. Available: <https://hive.blog/hive-174578/@timsaid/science-meets-photography-enhancing-image-brightness-with-python-and-numpy-a-deep-dive>. [Accessed 23 7 2024].
- [3] numpy, "numpy.clip," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.clip.html>. [Accessed 23 7 2024].
- [4] numpy, "numpy.flip," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.flip.html>. [Accessed 23 7 2024].
- [5] saturncloud, "How to Convert an Image to Grayscale Using NumPy Arrays: A Guide," 13 1 2024. [Online]. Available: <https://saturncloud.io/blog/how-to-convert-an-image-to-grayscale-using-numpy-arrays-a-comprehensive-guide/>. [Accessed 23 7 2024].
- [6] baeldung, "How to Convert an RGB Image to a Grayscale," 18 3 2024. [Online]. Available: <https://www.baeldung.com/cs/convert-rgb-to-grayscale>. [Accessed 23 7 2024].
- [7] dyclassroom, "How to convert a color image into sepia image," [Online]. Available: <https://dyclassroom.com/image-processing-project/how-to-convert-a-color-image-into-sepia-image>. [Accessed 23 7 2024].
- [8] wikipedia, "Kernel (image processing)," [Online]. Available: [https://en.wikipedia.org/wiki/Kernel\\_\(image\\_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)). [Accessed 25 7 2024].
- [9] numpy, "numpy.dot," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.dot.html>. [Accessed 23 7 2024].

- [10] numpy, "numpy.matmul," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.matmul.html#numpy.matmul>. [Accessed 23 7 2024].
- [11] numpy, "numpy.repeat," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.repeat.html>. [Accessed 23 7 2024].
- [12] numpy, "numpy.exp," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.exp.html>. [Accessed 25 7 2024].
- [13] numpy, "numpy.linspace," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.linspace.html>. [Accessed 25 7 2024].
- [14] numpy, "numpy.convolve," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.convolve.html>. [Accessed 25 7 2024].
- [15] wikipedia, "Gaussian blur," [Online]. Available: [https://en.wikipedia.org/wiki/Gaussian\\_blur](https://en.wikipedia.org/wiki/Gaussian_blur). [Accessed 25 7 2024].
- [16] M. L. stackexchange, "What is the rationale behind the "same" mode of discrete convolution?," [Online]. Available: <https://dsp.stackexchange.com/questions/89984/what-is-the-rationale-behind-the-same-mode-of-discrete-convolution>. [Accessed 25 7 2024].
- [17] wikipedia, "Convolution," [Online]. Available: [https://en.wikipedia.org/wiki/Convolution#Discrete\\_convolution](https://en.wikipedia.org/wiki/Convolution#Discrete_convolution). [Accessed 25 7 2024].
- [18] wikipedia, "Unsharp masking," [Online]. Available: [https://en.wikipedia.org/wiki/Unsharp\\_masking](https://en.wikipedia.org/wiki/Unsharp_masking). [Accessed 25 7 2024].
- [19] numpy, "numpy.ogrid," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.ogrid.html>. [Accessed 26 7 2024].
- [20] "Đồ thị hai dao động elip elip xiên đoạn thẳng," [Online]. Available: <https://123docz.net/document/10600084-do-thi-hai-dao-dong-elip-elip-xien-doan-thang.htm>. [Accessed 26 7 2024].
- [21] numpy, "numpy.fliplr," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.fliplr.html#numpy.fliplr>. [Accessed 26 7 2024].
- [22] wikipedia, "Bicubic interpolation," [Online]. Available: [https://en.wikipedia.org/wiki/Bicubic\\_interpolation#Bicubic\\_convolution\\_algorithm](https://en.wikipedia.org/wiki/Bicubic_interpolation#Bicubic_convolution_algorithm). [Accessed 29 7 2024].
- [23] numpy, "numpy.where," [Online]. Available: <https://numpy.org/doc/stable/reference/generated/numpy.where.html>. [Accessed 29 7 2024].