

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

-----o0o-----



BÁO CÁO ĐỒ ÁN LAB 2
IMAGE PROCESSING

<i>Thành viên</i>	Lê Phước Thạnh	22127392
<i>Lớp</i>	22CLC09	
<i>Học phần</i>	Toán ứng dụng và thống kê	MTH00051

Mục lục

I) Thông tin cá nhân:	3
II) Ý tưởng thuật toán và mô tả hàm:	3
1) Thay đổi độ sáng ảnh:	3
2) Thay đổi độ tương phản ảnh:	5
3) Lật ảnh ngang/dọc:	6
4) Chuyển đổi ảnh màu thành ảnh trắng đen/ảnh sepia:	7
5) Làm mờ/sắc nét ảnh:	9
6) Cắt ảnh theo kích thước:	13
7) Cắt ảnh theo khung tròn/khung elip:	14
8) Thay đổi kích thước ảnh:	17
9) Các hàm phụ:	20
III) Kiểm thử và đánh giá kết quả:	22
IV) Tài liệu tham khảo:	36

I) Thông tin cá nhân:

- Họ và tên: Lê Phước Thanh.

- MSSV: 22127392.

- Lớp: 22CLC09.

II) Ý tưởng thuật toán và mô tả hàm:

1) Thay đổi độ sáng ảnh:

a) Ý tưởng thuật toán:

- Ý tưởng chung của việc tăng/giảm độ sáng là đưa giá trị màu của ảnh về gần/xả giá trị màu trắng (255).

- Thuật toán sử dụng “hiệu chỉnh Gamma” (Gamma correction^[1]) để có thể tăng, giảm độ sáng của bức ảnh tùy thuộc vào hệ số Gamma γ .

- Vì mắt người cảm nhận độ sáng và màu sắc một cách phi tuyến tính và xấp xỉ hàm mũ (Explanation^[1]).

- Việc sử dụng hiệu chỉnh gamma sẽ cho ra kết quả ảnh được tăng độ sáng đồng đều và tạo cảm giác độ sáng giữa các điểm ảnh được mượt hơn.

- So với việc tăng giá trị màu một cách tuyến tính sẽ có khả năng làm thay đổi độ bảo hòa hoặc độ tương phản, khiến cho kết quả nhìn không được tự nhiên.

- Công thức tổng quát^[1]:

$$V_{out} = AV_{in}^{\gamma}$$

+ V_{out} : Giá trị đầu ra trong khoảng $[0,1]$.

+ V_{in} : Giá trị đầu vào trong khoảng $[0,1]$.

+ A : Hằng số (thường là 1).

+ γ : Hệ số Gamma.

- Đối với ảnh, giá trị đầu vào và đầu ra có giá trị $[0,255]$. Nên ta chia cả V_{out} và V_{in} cho 255. Khi đó ta có công thức hiệu chỉnh độ sáng ảnh^[2]:

$$O = \left(\frac{I}{255} \right)^{\gamma} \times 255$$

+ O : Giá trị của từng thành phần của điểm ảnh (RGB).

+ I : Giá trị RGB tương ứng của một điểm ảnh.

+ γ : Hệ số Gamma.

- Trong công thức trên, độ sáng sẽ tăng khi $\gamma < 1$, và độ sáng sẽ giảm khi $\gamma > 1$.

b) Mô tả hàm:

- Tham số đầu vào:

+ `img_2d`: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

+ `gamma`: số thực đại diện hệ số gamma. Độ sáng sẽ tăng khi gamma càng nhỏ ($\gamma < 1$) và ngược lại. Nếu $\gamma = 1$ thì sẽ không có thay đổi.

- Giá trị đầu ra:

+ `new_img`: một ma trận 2D chứa các giá trị pixel sau khi thay đổi độ sáng.

- Mã giải:

+ Bước 1: Áp dụng công thức hiệu chỉnh Gamma. Thư viện numpy sẽ tự động nhân các giá trị thành phần của ma trận pixel tương ứng cho các giá trị số thực.

+ Bước 2: Điều chỉnh lại giới hạn giá trị các pixel đảm bảo các giá trị nằm trong khoảng $[0, 255]$. Sử dụng hàm `np.clip(<giá trị đang xét>, <min>, <max>)` để đảm bảo quy tắc trên.

2) Thay đổi độ tương phản ảnh:

a) Ý tưởng thuật toán:

- Ý tưởng chung của việc tăng/giảm độ tương phản là làm giảm/tăng độ sáng một màu tối và tăng/giảm độ sáng của một màu sáng. Hay tổng quát hơn là tăng/giảm độ chênh lệch giá trị giữa các điểm ảnh.
- Thuật toán được cài đặt đơn giản nhân tất cả giá trị điểm ảnh cho cùng một số n để có thể làm tăng/giảm độ chênh lệch các điểm ảnh đi n lần (Brightness and contrast adjustments^[2]):

$$\Delta = |p_1 - p_2| \rightarrow \alpha \Delta = |\alpha p_1 - \alpha p_2|$$

- + Δ : Độ chênh lệch giá trị.
- + p_1, p_2 : Giá trị điểm ảnh.
- + α : Hệ số tăng/giảm độ tương phản.

b) Mô tả hàm:

- Tham số đầu vào:
 - + `img_2d`: một ma trận 2D chứa các giá trị pixel (ảnh gốc).
 - + `alpha`: số thực đại diện hệ số alpha. Độ sáng sẽ tăng khi alpha càng tăng và ngược lại ($\alpha > 0$). Nếu $\alpha = 1$ thì sẽ không có thay đổi.
- Giá trị đầu ra:
 - + `new_img`: một ma trận 2D chứa các giá trị pixel sau khi thay đổi độ tương phản.
- Mã giải:
 - + Bước 1: Nhân các giá trị màu của pixel cho hệ số α . Thư viện numpy sẽ tự động nhân các giá trị thành phần của ma trận pixel tương ứng cho các giá trị số thực.
 - + Bước 2: Điều chỉnh lại giới hạn giá trị các pixel đảm bảo các giá trị nằm trong khoảng $[0, 255]$. Sử dụng hàm `np.clip(<giá trị đang xét>, <min>, <max>)` để đảm bảo quy tắc trên.

3) Lật ảnh ngang/dọc:

a) Ý tưởng thuật toán:

- Ý tưởng chung của thuật toán là đảo ngược thứ tự các dòng/cột tương ứng với chiều ảnh cần lật.

- Với ảnh cần lật theo chiều ngang (trái phải), ta chỉ cần đổi chỗ các cột điểm ảnh trái sang cột phải:

$$\text{Swap}(c_i, c_{\text{size}-i}) \quad \forall i \in [0, \text{size} - 1]$$

+ c_i : Cột pixel i của ảnh.

- Với ảnh cần lật theo chiều dọc (trên dưới), ta chỉ cần đổi chỗ các dòng điểm ảnh trên sang dòng dưới:

$$\text{Swap}(r_i, r_{\text{size}-i}) \quad \forall i \in [0, \text{size} - 1]$$

+ r_i : Dòng pixel i của ảnh.

- Để tăng tốc độ của việc đổi chỗ, thuật toán sử dụng phương pháp slicing.

b) Mô tả hàm:

- Tham số đầu vào:

+ `img_2d`: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

+ `direction`: chuỗi ký tự đại diện cho hướng lật ảnh. Để lật ảnh trái-phải: `direction = 'horizontal'`; trên-dưới: `direction = 'vertical'`, cả hai hướng cùng lúc: `direction = 'both'`.

- Giá trị đầu ra:

+ `new_img`: một ma trận 2D, ảnh sau khi lật theo hướng đã chọn.

- Mã giải:

+ Bước 1: Kiểm tra hướng cần lật.

+ Bước 2: Dùng phương pháp slicing để hoán đổi vị trí các dòng/cột tương ứng theo hướng lật đã chọn. Để dùng phương pháp slicing để hoán đổi ta sử dụng hàm `np.fliplr()` để lật trái-phải và `np.flipud()` để lật trên-dưới được hỗ trợ trong `numpy`.

4) Chuyển đổi ảnh màu thành ảnh trắng đen/ảnh sepia:

a) Ý tưởng thuật toán:

- Ý tưởng chung của việc đưa ảnh màu về ảnh trắng/đen hay ảnh sepia là biến đổi các giá trị màu RGB của pixel thành các màu trắng/đen hay tông màu sepia tương ứng.
- Với màu trắng đen ta có thể đơn giản lấy giá trị trung bình các giá trị RGBs của điểm ảnh^{[3] [4]}:

$$\text{Gray} = (\text{Red} + \text{Green} + \text{Blue})/3.$$

- Tuy nhiên, vì mắt người cảm nhận 3 màu không đồng đều nhau. Cụ thể, mắt người nhạy cảm nhất với màu xanh lá và ít nhạy cảm nhất với màu xanh dương^[5]. Nên để hình ảnh được tự nhiên và giữ được các chi tiết, màu xám sẽ được tính bằng công thức dựa trên Luma^[6]:

$$Y' = a_1 R_{linear} + a_2 G_{linear} + a_3 B_{linear}$$

+ Y' : Giá trị màu xám.

+ a_1, a_2, a_3 : Hệ số Luminance (hệ số cảm nhận màu sắc).

+ $R_{linear}, G_{linear}, B_{linear}$: Giá trị RGB.

- Thuật toán sẽ sử dụng hệ số a theo chuẩn ITU-R BT.2100^[7] dành cho HDR^[8], đảm bảo chi tiết của những ảnh có màu chủ đạo sáng hay tối vẫn được đảm bảo:

$$a = (0.2627, 0.6780, 0.0593). \text{ [6]}$$

- Với màu sepia, phương pháp tương tự cũng được áp dụng, theo tổng hợp nhiều nguồn, theo kinh nghiệm và cảm nhận. Màu sepia được tạo ra bằng cách nhân vector màu RGB của pixel với ma trận với các hệ số^{[9] [10] [11] [12] [13]}:

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

Hay

$$\begin{aligned} \text{newRed} &= 0.393 * R + 0.769 * G + 0.189 * B \\ \text{newGreen} &= 0.349 * R + 0.686 * G + 0.168 * B \\ \text{newBlue} &= 0.272 * R + 0.534 * G + 0.131 * B \end{aligned}$$

b) Mô tả hàm:

- Tham số đầu vào:

+ `img_2d`: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

- Giá trị đầu ra:

+ `new_img`: một ma trận 2D, ảnh sau khi chuyển đổi sang trắng/đen hoặc sepia theo hướng đã chọn.

- Mã giải:

+ Bước 1:

- Với màu trắng đen: thực hiện phép nhân vô hướng tất cả các pixel với vector hệ số như đã nêu. Sử dụng `np.tensordot(<mảng1>,<mảng2>,<chiều thực hiện thao tác>)`. Để tất cả các pixel đều được nhân tích vô hướng ta sẽ khai báo chiều thao tác `axis = [2][0]` với `[2]` là chiều dữ liệu của màu và `[0]` là dòng 0 của chiều dữ liệu màu.
- Với màu sepia: thực hiện phép nhân ma trận giữa các vector màu và ma trận hệ số. Sử dụng phép nhân `@` được numpy hỗ trợ và phương pháp slicing.
 - Trong sepia có cú pháp `[...,:3]`: dấu ‘...’ có nghĩa là slicing các chiều trước chiều của màu (kích thước) hay `[...,:3] =[:,::,3]`.

+ Bước 2: Điều chỉnh lại giới hạn giá trị các pixel đảm bảo các giá trị nằm trong khoảng `[0,255]`. Sử dụng hàm `np.clip(<giá trị đang xét>,<min>,<max>)` để đảm bảo quy tắc trên.

5) Làm mờ/sắc nét ảnh:

a) Ý tưởng thuật toán:

- Với việc làm mờ ảnh:

+ Để làm mờ một điểm ảnh, ta có thể phân chia giá trị của điểm ảnh sang các điểm ảnh xung quanh. Để phân chia giá trị điểm ảnh, ta có thể nhân ma trận là tập hợp các điểm xung quanh điểm ảnh cần làm mờ với một ma trận kernel có cùng kích thước^[14]. VD: Trích từ Wikipedia. ^[14]

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right) [2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

Với [2,2] là tọa độ điểm ảnh cần làm mờ.

+ Có nhiều loại kernel khác nhau, nhưng để đảm bảo các cạnh, rìa của ảnh được rõ nét hơn (đảm bảo giữ được các chi tiết chính). Kernel được áp dụng là **Gauss kernel**. Với công thức: ^[15]

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

Hay

$$G(x, y) = \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

+ x,y: tọa độ của các phần tử trong kernel (tính từ phần tử trung tâm).

+ σ : độ mờ mong muốn.

+ Đối với những điểm ảnh ở phần rìa của ảnh, vì các điểm ảnh xung quanh không đủ để đảm bảo đúng kích thước với kernel. Để giải quyết việc đó, ta sẽ thêm các điểm ảnh phụ vào xung quanh khi làm mờ các điểm ngoài rìa. Hay còn gọi là padding.

+ Để giảm thời gian tính toán các giá trị. Thay vì dùng vòng lặp for thì thuật toán sẽ tạo ra một ‘góc nhìn’ hay một ma trận tạm thời có kích thước bằng với kích thước ảnh gốc nhưng mỗi phần tử sẽ là ma trận tập hợp các điểm ảnh xung quanh điểm ảnh đang xét.

VD: Với kernel có kích thước = 3 thì ma trận mới A’ và các padding = 0

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow A' = \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 4 & 5 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 2 & 3 & 0 \\ 5 & 6 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 & 2 \\ 0 & 4 & 5 \\ 0 & 7 & 8 \end{bmatrix} & \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} & \begin{bmatrix} 2 & 3 & 0 \\ 5 & 6 & 0 \\ 8 & 9 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 4 & 5 \\ 0 & 7 & 8 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 5 & 6 & 0 \\ 8 & 9 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix}$$

+ Với ma trận mới này, ta chỉ cần tận dụng broadcast các phép tính của numpy để tính các giá trị làm mờ nhanh chóng hơn.

- Với việc làm sắc nét ảnh:

+ Ý tưởng chung cũng tương tự như cách làm mờ, ta sẽ có một kernel riêng để làm sắc nét.

+ Tuy nhiên ta cũng có thể áp dụng công thức biến đổi unsharp masking: [\[16\]](#)

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

+ Với amount là hệ số làm rõ nét.

b) Mô tả hàm:

- Hàm làm mờ ảnh:

- Tham số đầu vào:

+ img_2d: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

+ size: số nguyên dương lẻ đại diện kích thước của kernel. Size càng lớn ảnh càng mờ

+ sigma: số thực đại diện hệ số làm mờ trong hàm phân phối chuẩn Gauss, số càng lớn, ảnh càng mờ. Mặc định = 1.0

- Giá trị đầu ra:

+ Ma trận 2D, ảnh sau khi làm mờ

- Mã giải:

+ Bước 1: Xây dựng ma trận Gauss kernel.

- Tạo một hệ trục tọa độ theo kích thước kernel đã chọn bằng np.linspace(); tạo trục Ox, Oy; và np.meshgrid() để tạo hệ trục từ 2 trục vừa tạo.

- Tính các giá trị trong kernel dựa vào công thức phân phối chuẩn Gauss đã nêu ở ý tưởng thông qua hàm `np.exp()`.
- Chuẩn hóa bằng cách chia từng phần tử trong kernel với tổng tất cả các phần tử trong kernel (Implementation [\[15\]](#)). Thông qua `np.sum()`.

+ Bước 2: Tạo padding cho các điểm ảnh ngoài rìa dựa vào kích thước của kernel. Dùng hàm `np.pad(<mảng gốc>,<số padding vào chiều dữ liệu>,<mode=<dữ liệu điền vào padding>)`. Để không phải thêm các giá trị ngoại lai vào padding, ta dùng `mode = 'reflect'` để các phần tử padding sẽ được lấy đối xứng qua điểm ảnh rìa đang xét. Tùy vào loại ảnh là RGB hay trắng đen mà số padding cũng như chiều dữ liệu sẽ tương ứng.

+ Bước 3: Như đã nói ở trên, để tăng tốc độ tính toán, ta sẽ tạo một ma trận tạm để có thể tận dụng sự broadcast của numpy để tính nhanh các giá trị. Để tạo ma trận tạm ta sẽ dùng thư viện phụ `stride` để thao tác trên byte của numpy cụ thể là **`np.lib.stride_tricks.asstrided()`**

+ Về bản chất thì ma trận tạm tạo bởi hàm **`asstrided(<mảng gốc>,<shape = <kích thước các chiều mới>,<strides><thông số để di chuyển giữa các phần tử mảng gốc>)`** là một ma trận ảo. Mỗi phần tử trong ma trận này sẽ được tra cứu hay ánh xạ qua ma trận gốc. Biến **`shape`** sẽ định hình kích thước các chiều của ma trận ảo, và biến **`strides`** là cách để truy xuất giá trị trong ma trận gốc theo một chiều nhất định.

+ `stride` đơn giản là số lượng byte để đi đến phần tử kế tiếp trong mảng theo một chiều nào đó.

VD với ma trận A và ma trận ảo A':

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow A' = \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 4 & 5 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 2 & 3 & 0 \\ 5 & 6 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 & 2 \\ 0 & 4 & 5 \\ 0 & 7 & 8 \end{bmatrix} & \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} & \begin{bmatrix} 2 & 3 & 0 \\ 5 & 6 & 0 \\ 8 & 9 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 4 & 5 \\ 0 & 7 & 8 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 5 & 6 & 0 \\ 8 & 9 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix}$$

+ Khi này `A.shape() = (3,3)` và `A.stride() = (3,1)` (trong đó giữa 2 phần tử kế tiếp trong dòng thì cách nhau 1 byte, và các phần tử kế nhau trong cột cách nhau số cột*khoảng cách byte). Và `A'.stride() = (3,1,3,1)`, với 3 đầu tiên

là số byte cần đi trong ma trận gốc để đến được phần tử tương ứng trong 3 dòng ma trận 3x3 của A', tương tự với số 1 thứ 2. Với số 3 thứ 3 thì đó là số byte cần đi trong ma trận gốc để đến các phần tử trong nội bộ một ma trận con 3x3 của ma trận A'.

+ Tương tự như vậy đối với ảnh màu RGB. Ta sẽ xây dựng được ma trận ảo như mong muốn.

+ Bước 4: Cuối cùng ta dùng hàm **np.enisump**(subscript=<phép tính mong muốn>, mảng 1, mảng 2) để nhân vô hướng các ma trận pixel con của ma trận ảo với ma trận kernel từ đó thu được ảnh mờ. subscript sẽ thực hiện các thao tính toán để đưa từ 2 ma trận ban đầu thành một ma trận với chiều dữ liệu mong muốn. VD: với ảnh trắng đen: subscript = 'ijkl, kl->ij'; từ ma trận ảo có shape = (cao, rộng, kích thước kernel, kích thước kernel) và ma trận kernel có shape = (kích thước kernel, kích thước kernel) thì kết quả sẽ là ma trận có shape = (cao, rộng). Và các giá trị của kết quả sẽ là phép nhân vô hướng của các ma trận con trong ma trận ảo với kernel. Hàm **np.enisump()** hoạt động dựa trên **Einstein notation**^[17].

- Hàm làm sắc nét ảnh:

- Tham số đầu vào:

+ img_2d: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

+ size: số nguyên dương lẻ đại diện kích thước của kernel. Size càng lớn ảnh rõ.

+ alpha: số thực đại diện hệ số làm sắc nét trong hàm unsharp masking. Alpha càng lớn, ảnh càng sắc nét

- Giá trị đầu ra:

+ sharpen_img: Ma trận 2D, ảnh sau khi làm sắc nét.

- Mã giải:

+ Bước 1: Tạo ảnh mờ Gauss blur bằng hàm đã cài đặt với sigma mặc định = 1.0 và kích thước kernel = size.

+ Bước 2: Áp dụng công thức:

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

+ Bước 3: : Điều chỉnh lại giới hạn giá trị các pixel đảm bảo các giá trị nằm trong khoảng $[0,255]$. Sử dụng hàm `np.clip(<giá trị đang xét>,<min>,<max>)` để đảm bảo quy tắc trên.

6) Cắt ảnh theo kích thước:

a) Ý tưởng thuật toán:

- Xác định vị trí điểm khởi đầu và kết thúc dựa vào kích thước cần cắt.
- Dựa vào vị trí tìm được để dùng slicing, tạo một ma trận mới.

b) Mô tả hàm:

- Tham số đầu vào:

+ `img_2d`: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

+ `size`: số nguyên đại diện kích thước ảnh cần cắt.

- Giá trị đầu ra:

+ `new_img`: một ma trận 2D với kích cỡ `size*size`.

- Mã giải:

+ Bước 1: Tìm dòng và cột khởi đầu. Vì hình cần cắt nằm ở trung tâm ảnh gốc, nên ta áp dụng công thức:

$\text{Dòng/cột bắt đầu} = (\text{chiều dài/rộng-size})/2 \text{ lấy phần nguyên.}$

Suy ra: $\text{Dòng/cột kết thúc} = \text{Dòng/cột bắt đầu} + \text{size}.$

+ Bước 2: Dùng slicing tạo ma trận mới dựa trên dòng/cột tìm được.

7) Cắt ảnh theo khung tròn/khung elip:

a) Ý tưởng thuật toán:

- **Với khung tròn:** Tạo một ma trận bool có cùng kích cỡ với ma trận ảnh. Xác vị trí tâm hình tròn cũng là tâm của hình vuông. Với mỗi phần tử trong ma trận bool, xét khoảng cách của chúng tới tâm. Nếu khoảng cách lớn hơn bán kính thì điền False, ngược lại là True.

+ Sau đó áp dụng ma trận bool vừa có vào ma trận ảnh. Lúc đó, những điểm nằm ngoài bán kính sẽ bị tô đen.

- **Với khung elip:** Tạo một ma trận bool có hình dạng mong muốn và áp dụng ma trận đó như cách làm với khung tròn.

+ Để vẽ hình elip chéo, đầu tiên ta sẽ chuyển tọa độ các điểm ảnh thành tọa độ mới có hệ trục trùng với các trục của elip^[18]:

$$\begin{aligned} X &= (x - x_o) \cos \theta + (y - y_o) \sin \theta, \\ Y &= -(x - x_o) \sin \theta + (y - y_o) \cos \theta. \end{aligned}$$

+ X,Y: Tọa độ các điểm cũ trong hệ trục mới.

+ x_o, y_o : Tọa độ gốc của hệ trục mới.

+ θ : Góc xoay giữa hệ trục cũ và mới.

+ Sau đó ta áp dụng phương trình elip và điền False vào những điểm thỏa phương trình > 1 :

$$\frac{X^2}{a^2} + \frac{Y^2}{b^2} > 1$$

+ Giá trị của trục chính a và phụ b được xác định như sau:

+ Để hình elip chạm vào cạnh hình vuông tại một điểm ta chỉ cần giải phương trình hoành độ giao điểm của elip và cạnh đó sao cho chúng giao nhau đúng tại một điểm:

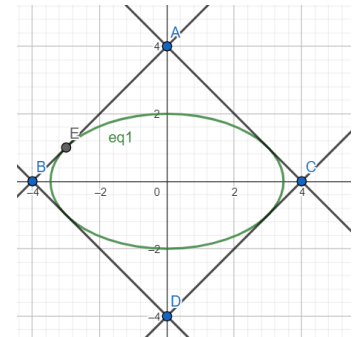
+ Giả sử ta lấy cạnh n có phương trình đường thẳng:

$$y = x + r \quad \text{với } r = \text{nửa độ dài đường chéo.}$$

+ Đặt hình elip có phương trình:

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

+ Thay phương trình đường thẳng vào y của elip:



$$\frac{x^2}{a^2} + \frac{(x+r)^2}{b^2} = 1.$$

$$\rightarrow \left(\frac{1}{a^2} + \frac{1}{b^2}\right)x^2 + \frac{2r}{b^2}x + \frac{r^2}{b^2} - 1 = 0$$

+ Để cả hai giao nhau tại 1 điểm thì phương trình trên có 1 nghiệm duy nhất.

$$\Delta = b^2 - 4ac = \frac{4(a^2 + b^2 - r^2)}{a^2 b^2} = 0$$

$$\rightarrow a^2 = r^2 - b^2 \quad (1)$$

+ Với r là nửa đường chéo nên $r = \frac{\sqrt{2}}{2}d$ với d = độ dài cạnh hình vuông.

$$+ \text{Đặt } b = nr = n \frac{\sqrt{2}}{2}d.$$

$$+ \text{Do đó từ (1)} \rightarrow a = \frac{\sqrt{2}}{2}d\sqrt{1 - n^2}$$

$$+ \text{Vì } a \geq b \rightarrow \sqrt{1 - n^2} \geq n^2 \rightarrow n \leq \frac{1}{\sqrt{2}}$$

$$+ \text{Để hình ảnh trong đẹp mắt thì hệ số n sẽ mặc định } = \frac{1}{2}.$$

b) Mô tả hàm:

- Hàm khung tròn:

- Tham số đầu vào:

+ img_2d: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

+ radius: số nguyên đại diện bán kính của hình tròn (đường kính không quá kích thước ảnh).

- Giá trị đầu ra:

+ new_img: một ma trận 2D với khung tròn có bán kính đã nhập.

- Mã giải:

+ Bước 1: Xác định tọa độ trung tâm.

+ Bước 2: Kiểm tra tính hợp lệ của bán kính.

+ Bước 3: Khởi tạo các trục tọa độ Ox, Oy. Hàm np.ogrid[,] sẽ trả về 1 vector chứa các dòng và 1 vector chứa cột đại diện cho 2 trục tọa độ.

+ Bước 4: Tận dụng việc broadcast khi tính toán với ma trận của numpy để có thể tính khoảng cách của từng tọa độ và kiểm tra chúng với bán kính để thu được ma trận bool trong ý tưởng.

+ Bước 5: Thực hiện phép nhân từng pixel với ma trận bool thông qua slicing.

- Hàm *khung elip*:

- Tham số đầu vào:

+ img_2d: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

+ thickness_coefficient: số thực đại diện hệ số giữa độ dài trục phụ so với độ dài nửa đường chéo. Nếu hệ số nhập vào khác $[0, \frac{1}{\sqrt{2}}]$ thì sẽ sử dụng hệ số mặc định là $\frac{1}{2}$.

- Giá trị đầu ra:

+ new_img: một ma trận 2D với khung 2 elip bắt chéo nhau với độ dày đã nhập.

- Mã giải:

+ Bước 1: Xác định tọa độ trung tâm.

+ Bước 2: Tính độ dài các trục chính a, trục phụ b của elip dựa trên hệ số đã nhập theo công thức đã chứng minh ở ý tưởng.

+ Bước 3: Khởi tạo các trục tọa độ Ox, Oy. Hàm np.ogrid[,] sẽ trả về 1 vector chứa các dòng và 1 vector chứa cột đại diện cho 2 trục tọa độ.

+ Bước 4: Tính góc xoay $= \frac{\pi}{4}$. Vì hình vuông có góc đường chéo $= 45^\circ$.

+ Bước 5: Đổi hệ trục tọa độ. Lấy đối xứng hệ trục để được hình elip số 2.

+ Bước 6: Tận dụng việc broadcast khi tính toán với ma trận của numpy để áp dụng phương trình elip và kiểm tra chúng với điều kiện thuộc elip để được ma trận bool của một hình elip. Tương tự tìm ma trận bool hình elip thứ 2. Dùng hàm np.logical_or() để gộp 2 ma trận elip thu được ma trận bool có hình như ý.

+ Bước 7: Thực hiện phép nhân từng pixel với ma trận bool thông qua slicing.

8) Thay đổi kích thước ảnh:

a) Ý tưởng thuật toán:

- Ý tưởng chung của bài toán thay đổi kích thước là tìm giá trị cho những điểm ảnh trên ảnh mới sao cho phù hợp với ảnh gốc. Một dạng bài toán tối ưu.
- Có rất nhiều phương pháp để tính các giá trị điểm ảnh mới như Nearest-neighbor interpolation, Bilinear, Bicubic, Fourier-transform,... Mỗi phương pháp đều có ưu nhược điểm riêng (càng bảo toàn tốt các chi tiết ảnh thì công thức tính toán và cài đặt càng khó) [\[19\]\[20\]\[21\]](#).
- Để cân bằng giữa độ hiệu Thuật toán được cài đặt dựa trên phương pháp Bicubic Interpolation (nội suy song khối) [\[22\]](#).
- Đầu tiên, xét một pixel $p_{i,j}$ tại vị trí bất kỳ trên ảnh mới. Ta sẽ đổi chiều vị trí $p_{i,j}$ với pixel $p_{m,n}$ trong ảnh gốc tại vị trí tương ứng gần nhất. Từ đó tìm được pixel gốc tương ứng mà pixel mới $p_{i,j}$ sẽ biểu diễn đại diện.
- Sau khi xác định được $p_{m,n}$, mà pixel $p_{i,j}$ đại diện ta cần xác định một **ma trận pixel 4x4 (gọi là patch)** gồm các pixel xung quanh $p_{m,n}$ (các pixel $p_{m+a,n+a}$ với $a \in [-1,2]$).
- Khi đó trọng số theo các trục tọa độ của $p_{i,j}$ sẽ được tính dựa vào hàm bậc ba (Cubic) đã được tính toán để hình thành kernel:

$$W(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1 & \text{for } |x| \leq 1, \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & \text{for } 1 < |x| < 2, \\ 0 & \text{otherwise,} \end{cases} \quad [22]$$

+ Với x là chênh lệch khoảng cách xét trên một trục (Ox, Oy) nhất định giữa $p_{i,j}$ và các điểm xung quanh $p_{m,n}$.

+ a : hệ số của kernel, thường lấy -0.5 hoặc -0.75 .

+ $W(x)$: trọng số theo một trục của pixel cần tìm ($W(x)$ là một vector R^4).

- Sau khi tính trọng số trên trục Ox và Oy đối với vị trí tất cả các điểm trên ảnh mới. Khi đó giá trị của pixel trong ảnh mới sẽ bằng:

$$P(t,p) = \sum_{i=-1}^2 \sum_{j=-1}^2 patch(x+i, y+j) * W_{Ox}(x+i) * W_{Oy}(y+j)$$

+ Với $P(t,p)$: giá trị pixel tại vị trí t,p trong ảnh mới.

+ patch: ma trận 4x4 tập hợp các điểm xung quanh điểm $P(x,y)$ trên ảnh gốc mà điểm $P(t,p)$ trên ảnh mới đại diện biểu diễn.

- + x, y : tọa độ của pixel trên ảnh gốc của mà $P(t,p)$ đại diện.
- + $W_{Ox}(x+i), W_{Oy}(y+j)$: trọng số chênh lệch khoảng cách giữa điểm $P(t,p)$ với các điểm xung quanh điểm mà nó đại diện theo trục Ox và Oy .
- Để tìm tọa độ pixel gốc mà pixel mới $p_{i,j}$ đại diện, ta chỉ cần lấy phần nguyên sau khi chia tọa độ (i,j) với tỷ lệ kích thước.
- Để tìm độ chênh lệch so với điểm gốc thì ta chỉ cần lấy phần thập phân sau khi chia (i,j) với tỷ lệ kích thước.
- Tương tự với độ lệch của điểm (i,j) so với các điểm xung quanh điểm gốc.
- Để giảm thiểu thời gian tính toán do dùng vòng lặp. Ta sẽ lại sử dụng phương pháp tạo ma trận ảo chứa các phần tử là tập hợp các **ma trận patch**, thông qua stride. Từ đó có thể tìm được các patch tương ứng đối với các pixel ở vị trí khác nhau trong ảnh mới.

b) Mô tả hàm:

- Hàm tính hệ số (hay kernel) theo Bicubic interpolation:

- Tham số đầu vào:

+ x : một mảng chứa độ chênh lệch khoảng cách của tất cả pixel $P_{i,j}$ mới theo một trục nhất định với tọa độ theo trục tương ứng của các pixel xung quanh pixel cũ mà pixel mới đại diện.

+ a : hệ số của phương trình Bicubic. Mặc định sử dụng là -0.5.

- Giá trị đầu ra:

+ Mảng có kích thước giống hệ x , với các phần tử tương ứng giờ là các trọng số để tìm giá trị.

- Mã giải:

+ Bước 1: Tận dụng việc broadcast khi tính toán với ma trận của numpy để tính toán các giá trị trọng số.

+ Bước 2: Kiểm tra và giữ lại những trọng số phù hợp với điều kiện áp dụng công thức.

- Hàm khung elip:

- Tham số đầu vào:

+ `img_2d`: một ma trận 2D chứa các giá trị pixel (ảnh gốc).

+ ratio: số thực đại diện tỷ lệ kích thước giữa ảnh mới và ảnh cũ. $\text{Ratio} = \text{kích thước ảnh mới} / \text{kích thước ảnh cũ}$.

- Giá trị đầu ra:

+ new_img: một ma trận 2D với kích thước đã thay đổi.

- Mã giải:

+ Bước 1: Xác định kích thước mới dựa vào tỷ lệ ratio.

+ Bước 2: Khởi tạo ảnh mới, và dựa vào tỷ lệ ratio để tính lại tọa độ của tất cả các pixel mới nếu đặt trong hệ tọa độ ảnh cũ. Hàm np.arange() dùng để tạo một mảng giống hàm range() nhưng của numpy.

+ Bước 3: Tìm tọa độ điểm gốc mà các pixel mới đại diện.

+ Bước 4: Tìm độ chênh lệch khoảng cách ở các trục Ox, Oy giữa pixel mới và pixel cũ. Đồng thời là độ chênh lệch giữa pixel mới với các pixel cũ xung quanh pixel cũ. Chỗ: np.arange(-1,3) – x/y_diff[:,None], là để tìm chênh lệch pixel mới với các điểm cũ xung quanh, lệnh này sẽ tạo ra ma trận với các cột là chênh lệch giữa pixel mới với pixel cũ i.

+ Bước 5: Áp dụng công thức tìm trọng số với các chênh lệch vừa tìm được.

+ Bước 6: Thêm padding cho ma trận cũ.

+ Bước 7: Áp dụng stride giống như ở làm mờ để thu được ma trận ảo kích thước (cao gốc*rộng gốc). Với mỗi phần tử trong ma trận là một **ma trận con patch (4x4)** chứa các pixel xung quanh pixel gốc ở vị trí tương ứng.

+ Bước 7: Xây dựng ma trận ảnh tạm có kích thước (cao mới*rộng mới) dựa trên ma trận ảo đã xây dựng, với mỗi phần tử tại (i,j) sẽ chứa ma trận patch của pixel cũ mà pixel mới (i,j) đại diện.

+ Bước 8: Áp dụng công thức tính giá trị điểm ảnh. Hàm np.einsum() sẽ được sử dụng giống như lúc làm mờ ảnh.

+ Bước 9: Điều chỉnh lại giới hạn giá trị các pixel đảm bảo các giá trị nằm trong khoảng [0,255]. Sử dụng hàm np.clip(<giá trị đang xét>,<min>,<max>) để đảm bảo quy tắc trên.

9) Các hàm phụ:

a) Hàm đọc ảnh:

- Tham số đầu vào:

+ `img_path`: đường dẫn đến file ảnh.

- Giá trị đầu ra:

+ `img_2d`: ma trận 2D với mỗi phần tử là một pixel.

- Mã giải:

+ Bước 1: Kiểm tra tính hợp lệ của đường dẫn file ảnh. Sử dụng thư viện `os` có sẵn trong python.

+ Bước 2: Mở và tải ảnh vào ma trận, đồng thời lấy các thông số của ảnh: kích thước dài rộng, số lượng kênh màu, tên file, loại file.

b) Hàm hiển thị ảnh:

- Tham số đầu vào:

+ `img`: ma trận 2D chứa các pixel.

- Thao tác: hiển thị ảnh với hàm `imshow()` của thư viện `matplotlib.pyplot`.

c) Hàm lưu ảnh:

- Tham số đầu vào:

+ `img`: ma trận 2D với mỗi phần tử là một pixel.

+ `img_path`: đường dẫn đến nơi lưu ảnh.

- Giá trị đầu ra:

+ `True/False` cho biết việc lưu thành công hay thất bại.

- Mã giải:

+ Bước 1: Kiểm tra tính hợp lệ của đường dẫn file ảnh. Sử dụng thư viện `os` có sẵn trong python.

+ Bước 2: Tạo ảnh với `Image.fromarray()` và lưu file ảnh với `Image.save()` của thư viện `PIL`.

d) Hàm main:

- Khởi đầu sẽ yêu cầu nhập ảnh cần xử lý.

- Tiếp theo chọn mode lưu ảnh. Nếu chọn 1 (có) thì tất cả các ảnh được xử lý sẽ được lưu vào output folder cho đến khi mode được chuyển sang 0 (không lưu).

- Người dùng sẽ nhập các lựa chọn tương ứng để thực hiện việc xử lý ảnh:

0. Thực hiện đồng thời tất cả các thao tác xử lý.

1. Thay đổi độ sáng.

2. Thay đổi độ tương phản.

3. Lật ảnh.

4. Chuyển đổi màu trắng/đen và màu sepia.

5. Làm mờ, sắc nét ảnh.

6. Cắt ảnh.

7. Cắt ảnh theo khung tròn/khung elip.

8. Thay đổi kích thước ảnh.

9. Thay đổi chế độ lưu (nếu muốn thay đổi vị trí lưu thì nhấn có thêm 1 lần nữa).

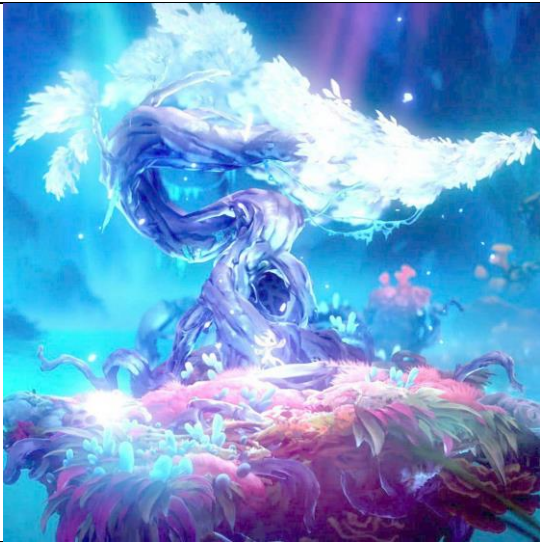

10. Thay đổi ảnh input.


- Chọn các lựa chọn khác để thoát chương trình.



III) Kiểm thử và đánh giá kết quả:



- Với ảnh gốc kích thước 1080x1080:








STT	Chức năng/hàm	Mức độ hoàn thành	Ảnh kết quả	Chú thích
1	Thay đổi độ sáng	100%		Gamma=0.5
2	Thay đổi độ tương phản	100%		Alpha=1.7

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%		Kernel size=5 Sigma = 3
5.2	Làm sắc nét ảnh	100%		Kernel size=5 Sigma=1 Alpha=3



6	Cắt ảnh theo kích thước	100%		Size=512
7.1	Cắt ảnh theo khung tròn	100%		Radius=512
7.2	Cắt ảnh theo khung elip	100%		Thickness = 0.5



8	Hàm main	100%		
9	Phóng to/thu nhỏ	100%		Ratio =0.2
				Ratio=2



- Với ảnh 2880x2880:








STT	Chức năng/hàm	Mức độ hoàn thành	Ảnh kết quả	Chú thích
1	Thay đổi độ sáng	100%		Gamma=0.5
2	Thay đổi độ tương phản	100%		Alpha=1.7

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%		Kernel size=13 Sigma = 13 (Sấp xỉ 10s)
5.2	Làm sắc nét ảnh	100%		Kernel size=7 Sigma=1 Alpha=13 (Sấp xỉ 7s)

6	Cắt ảnh theo kích thước	100%		Size=1440
7.1	Cắt ảnh theo khung tròn	100%		Radius=1440
7.2	Cắt ảnh theo khung elip	100%		Thickness = 0.5

8	Hàm main	100%		
9	Phóng to/thu nhỏ	100%		Ratio = 0.2
				Ratio=2

IV) Tài liệu tham khảo:

- Tài liệu:

Độ sáng và tương phản

[1].[Gramma Correction, Wikipedia](#) (truy cập 19/07/2024).

[2].https://docs.opencv.org/4.x/d3/dc1/tutorial_basic_linear_transform.html (truy cập 19/07/2024).

Ảnh trắng/đen và sepia

[3].<https://medium.com/@mjbharmal2002/gray-scaling-with-the-algorithms-b83f87975885> (truy cập 19/07/2024).

[4].<https://tannerhelland.com/2011/10/01/grayscale-image-algorithm-vb6.html> (truy cập 19/07/2024).

[5].[Grayscale, Wikipedia](#) (truy cập 19/07/2024).

[6]. [Luma \(video\), Wikipedia](#) (truy cập 19/07/2024).

[7]. [ITU-R BT.2100](#) (truy cập 19/07/2024).

[8].[HRD](#) (truy cập 19/07/2024).

[9].<https://www.geeksforgeeks.org/image-processing-in-java-colored-image-to-sepia-image-conversion/> (truy cập 19/07/2024).

[10].<https://stackoverflow.com/questions/1061093/how-is-a-sepia-tone-created> (truy cập 19/07/2024).

[11]. <https://leware.net/photo/blogSepia.html> (truy cập 19/07/2024).

[12].<https://stackoverflow.com/questions/9448478/what-is-wrong-with-this-sepia-tone-conversion-algorithm/9448635> (truy cập 19/07/2024).

[13].<https://abhijitnathwani.github.io/blog/2018/01/08/colortosepia-Image-using-C> (truy cập 19/07/2024).

Làm mờ/sắc nét ảnh

[14].[Kernel \(image processing\)](#) (truy cập 21/07/2024).

[15].[Gaussian blur](#) (truy cập 21/07/2024).

[16].[Unsharp Masking](#) (truy cập 21/07/2024).

[17].[Einstein notation](#) (truy cập 21/07/2024).

Vẽ khung

[18].[Ellipse, Wikipedia](#) (truy cập 22/07/2024).

Thay đổi kích thước:

[19].[Image Scaling, Wikipedia](#) (truy cập 28/07/2024).

[20].<https://viblo.asia/p/upscale-anh-voi-mot-mang-cnn-don-gian-Az45b04gZxY>
(truy cập ngày 28/07/2024).

[21].[So sánh các phương pháp thay đổi kích thước, Wikipedia](#) (truy cập 28/07/2024).

[22].[Bicubic Interpolation, Wikipedia](#) (truy cập 28/07/2024).

- Acknowledgement:

ChatGPT-4o.

Google Gemini.