VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

Applied Mathematics and Statistics

Project 02

Image Processing with NumPy

Student: 21127135 - Diep Huu Phuc
Class: 21CLC05
Instructors: Vu Quoc Hoang
Nguyen Van Quang Huy
Le Thanh Tung
Phan Thi Phuong Uyen

Ho Chi Minh City, July 2023

CONTENTS

1 INTRODUCTION **2**

2 IDEAS AND IMPLEMENTATIONS **2**
2.1 1, 2, 3 - Brightness, Contrast, Flipping 3
2.1.1 Ideas 3
2.1.2 Implementations 3
2.2 4 - Grayscale and Sepia 4
2.2.1 Ideas 4
2.2.2 Implementations 4
2.3 5 - Blurring and Sharpening 4
2.3.1 Ideas 4
2.3.2 Implementations 5
2.4 6, 7 - Center and Circle cropping 6
2.4.1 Ideas 6
2.4.2 Implementations 6
2.5 Extra - Cross Ellipses cropping 7
2.5.1 Ideas 7
2.5.2 Implementations 7

3 RESULTS AND DISCUSSION **10**
3.1 1 - Brightness 10
3.2 2 - Contrast 10
3.3 3 - Flipping 11
3.4 4 - Grayscale and Sepia 11
3.5 5 - Blurring and Sharpening 12
3.6 6, 7 - Center and Circle cropping 14
3.7 Extra - Cross Ellipses cropping 14

Project 02: Image Processing with NumPy

Diep Huu Phuc [1⋆]
[1] *Faculty of Information Technology, VNUHCM-University of Science, Vietnam*
⋆ *Student ID: 21127135, Github: kru01*

Abstract

A digital image when stripped to its core is just a matrix of pixels. Hinging on the kind of image, a pixel can take various shapes. Grayscale images, for instance, have every of their pixels down as a single value denoting its light intensity. On the other hand, the most well-known color model *RGB* utilizes a vector of numbers for its pixels, each figure can range from 0 to 255 which signifies the contribution's magnitude of either Red, Green, or Blue to the color of said pixel. Because all substantial elements of digital images are in numerical forms, mathematical concepts and tools can thus be wielded to process them. Digital image processing involves modifying the nature of an image to cater to situational demands. In this document, for a few selected rudimentary algorithms, central principles as well as requisite theories for successful implementations will be presented.

1 INTRODUCTION

Digital image processing refers to the act of processing digital images through an algorithm by employing a digital computer. It triumphs over **analog image processing** in many aspects, such as, allowing the input data to be applied with more diverse algorithms, and eliminating the noise and distortion build-up, etc. (Wikipedia contributors, 2023b). A general image processing operator is a function that takes one or more input images and produces an output image. Image transforms can be seen as: Point operators (pixel transforms) or Neighborhood (area-based) operators (OpenCV, 2023). Therefore, by reading an image as a `numpy.ndarray`, various processing operations can be performed using *NumPy* functions (Harris et al., 2020).

In this project, I was tasked with writing 8 image processing algorithms while only using *NumPy* to perform all the core calculations. Before going in-depth in Section 2, here is a table summarizing everything I have achieved.

Priority	No.	Task	Status (%)
Required	1, 2, 3	Brightness, Contrast, Flipping	100
	4	Grayscale and Sepia	100
	5	Blurring and Sharpening	100
	6, 7	Center and Circle cropping	100
	8	Main function	100
Extra	9	Cross Ellipses cropping	100

From this point onward,

- Every reference to `numpy` will be shortened to `np`.
- Functions with parameters will be written as `func(.)`.

2 IDEAS AND IMPLEMENTATIONS

There are 3 noteworthy details that should always be taken into consideration when creating all the functions.

- Because we exclusively deal with the *RGB* color model, it is essential that the input image interpreted as a `np.ndarray` has a shape of `(height, width, num_channels)`.
- When the value of a color channel is altered, we must ensure that the result stays in the range of $[0, 255]$.
- The output image returned by the function should consistently be of *unsigned 8-bit integer* datatype. *Uint*8 only allows whole numbers in $[0, 255]$, which also matches the range of a *RGB* color channel.

The mentioned specifics can be effortlessly tackled with `np.clip(.)` and `np.ndarray.astype('uint8')` respectively.

2.1 1, 2, 3 - Brightness, Contrast, Flipping

2.1.1 Ideas

Brightness adjustment is as simple as adding or substracting the desired change in brightness to each of the red, green and blue color components (Loch, 2010). However, a better way would be to multiply a brightness factor with 255 before adding it to the channels (Patrick, 2014).

$$channel_{brightness} = channel + (255 \cdot factor_{brightness}) \tag{1}$$

This equation also makes for more intuitive input since, instead of entering random integers, we can pinpoint exactly how darker or brighter the image should be in percentage.

Contrast is the difference between the maximum and minimum pixel intensity in an image (Point, 2023). Even so, solely modifying the values of these two pixels will not produce any difference. First, we have to compute a contrast correction factor before applying it to the color components (Loch, 2015). Furthermore, just as with **Brightness** adjustment, we fancy the input to be a factor.

$$\begin{aligned} contrast &= 255 \cdot factor_{contrast} \\ factor_{correction} &= \frac{259 \cdot (255 + contrast)}{255 \cdot (259 - contrast)} \\ channel_{contrast} &= factor_{correction} \cdot (channel - 128) + 128 \end{aligned} \tag{2}$$

Flipping can be quickly explained with just a simple visualization. To make it easier on the eyes, we can replace every $[R\ G\ B]$ array with a letter from the English alphabet.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \xrightarrow[\text{horizontally}]{\text{Flip}} \begin{bmatrix} c & b & a \\ f & e & d \\ i & h & g \end{bmatrix}, \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \xrightarrow[\text{vertically}]{\text{Flip}} \begin{bmatrix} g & h & i \\ d & e & f \\ a & b & c \end{bmatrix} \tag{3}$$

Nevertheless programmatically, this could become pretty complex. Fortunately, as we will explore in Section 2.1.2, the use of *NumPy* trivializes this problem.

2.1.2 Implementations

The equations 1 and 2 presented in Section 2.1.1 are already clear indications of the code's form for **Brightness** and **Contrast** adjustment. Onto **Flipping**, there exist mutiple handy functions that carry out precisely what were illustrated in Eq. 3, namely, `np.flip(.)`, `np.flipud(.)` and `np.fliplr(.)`.

2.2 4 - Grayscale and Sepia

2.2.1 Ideas

There are 3 methods to **grayscaling** an image, but **Lightness** holds one serious flaw so we will only look into the other two. **Average** takes the average value of the three components (red, green, blue) as the grayscale value. It is also problematic since each component is assigned the same weight, yet, our eyes are more sensitive to green, red, and blue accordingly. **Luminosity** comes in to solve said problem by calculating a weighted average. Blue's contribution to the final value should decrease while green's should increase (Antoniadis, 2023).

$$gray_{avg} = \frac{R+G+B}{3} \tag{4}$$

$$gray_{lum} = 0.3 \cdot R + 0.59 \cdot G + 0.11 \cdot B \tag{5}$$

Similarly, converting an image to **Sepia** is the process of computing a weighted average, except this time, it is for each color channel of the pixel (Shakeel, 2014).

$$\begin{aligned} R_{sep} &= 0.393 \cdot R + 0.769 \cdot G + 0.189 \cdot B \\ G_{sep} &= 0.349 \cdot R + 0.686 \cdot G + 0.168 \cdot B \\ B_{sep} &= 0.272 \cdot R + 0.534 \cdot G + 0.131 \cdot B \end{aligned} \tag{6}$$

2.2.2 Implementations

Average grayscaling with Eq. 4 can be fully conveyed through `np.mean(img, axis=2)`. `axis=2` alludes to the $2 + 1 = 3^{rd}$ entry in the `(height, width, num_channels)` shape, meaning we are working in the axis of the color channels. The function will replace every pixel in `img`, which is a $[R\ G\ B]$ matrix, with the average of its components.

As for **Luminosity grayscaling**, Eq. 5 can be rearranged to become the dot product of $p = [R\ G\ B]$ and the transpose of $w = [0.3\ 0.59\ 0.11]$. Then, iteratively dotting every single p with w^T can be condensed to computing the matrix multiplication of `img` and w^T. This is accomplished with `np.matmul(img, w)`, we don't even need to use w^T since w is an 1-D array and will be implicitly handled by *NumPy* (tel, 2021).

Following the same reasoning as **Luminosity**, but with 3 separate weights this time,

$$w_R = [0.393\ 0.769\ 0.189], w_G = [0.349\ 0.686\ 0.168], w_B = [0.272\ 0.534\ 0.131]$$

we proceed to work out an individual array for each channel, `Xs = np.matmul(img,`w_X`)`, before forming the output image with `np.dstack((Rs,Gs,Bs))` (denis, 2012).

2.3 5 - Blurring and Sharpening

2.3.1 Ideas

In image processing, a **kernel**, or **mask** is a small matrix used to apply a particular filter on an image by doing a **convolution** between itself and the image. Thankfully, in the confines of this project, the available pre-computed kernels are more than enough.

$$Sharpen = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}, Blur_{Gaussian3x3} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{7}$$

So the only thing needed to be addressed is the **convolve** algorithm. **Convolution** is the process of adding each element of the image to its local neighbors, weighted by the kernel (Wikipedia contributors, 2023a). Additionally, a decision should be made regarding what should happen when edges are reached. We could wrap the image around or duplicate the edges, etc., but for simplicity, I chose to ignore the parts of the kernel that had strayed out of the image and did a slightly smaller averaging for the corners. That means the blurring around the edges would be faintly diminished, still, this probably won't make any noticeable difference (Pound, 2015).

2.3.2 Implementations

- My `convole(.)` function supports both the *RGB* color scheme and images with one-value pixels, e.g., **grayscaled** and **Sepia** images. With that said, we will only be covering working with *RGB* since that is much more comprehensive.

First step taken in our `convolve(img, kernel)` function is padding the outer of the image array with 0s so when the kernel inevitably gets there, it won't produce any errors.

$$\begin{bmatrix} RGB_{11} & RGB_{12} & \dots & RGB_{1N} \\ RGB_{21} & RGB_{22} & \dots & RGB_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ RGB_{K1} & RGB_{K2} & \dots & RGB_{KN} \end{bmatrix} \xrightarrow{\text{Padding}} \begin{bmatrix} 000 & 000 & \dots & 000 & 000 \\ 000 & RGB_{11} & \dots & RGB_{1N} & 000 \\ 000 & RGB_{21} & \dots & RGB_{2N} & 000 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 000 & RGB_{K1} & \dots & RGB_{KN} & 000 \\ 000 & 000 & \dots & 000 & 000 \end{bmatrix}$$

After failing to get `np.pad(.)` to construct the desired matrix, I went through *GitHub* and found a clever method of using `np.zeros(.)` to initialize an array with `(height + 2, width + 2, num_channels)` shape, i.e., having 2 more rows and columns compared to the input image. Next, **slicing** is employed to nestle the original image array into the newly created one (pooya kalahroodi, 2023).

"Weighted by the kernel" from the **convolution** process specified in Section 2.3.1 is a form of element-wise matrix multiplication, called **Hadamard product**, of the kernel and the *sub-matrix* formed by projecting said kernel on a pixel of the image array. This product can be determined with `np.multiply(.)` (P, 2018).

$$\begin{aligned} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \overset{Hadamard}{\times} \begin{bmatrix} RGB_{11} & RGB_{12} & RGB_{13} \\ RGB_{21} & RGB_{22} & RGB_{23} \\ RGB_{31} & RGB_{32} & RGB_{33} \end{bmatrix} \\ = \begin{bmatrix} 000_{11} & -1 \cdot RGB_{12} & 000_{13} \\ -1 \cdot RGB_{21} & 5 \cdot RGB_{22} & -1 \cdot RGB_{23} \\ 000_{31} & -1 \cdot RGB_{32} & 000_{33} \end{bmatrix} \end{aligned} \tag{8}$$

Although, leaving our kernels as they are in Eq. 7 wouldn't provide us with the correct result in Eq. 8. Instead, we would get

$$\begin{bmatrix} \begin{bmatrix} 0 & -1 \cdot G & 0 \end{bmatrix}_{11} & \begin{bmatrix} -1 \cdot R & 5 \cdot G & -1 \cdot B \end{bmatrix}_{12} & \begin{bmatrix} 0 & -1 \cdot G & 0 \end{bmatrix}_{13} \\ \begin{bmatrix} 0 & -1 \cdot G & 0 \end{bmatrix}_{21} & \begin{bmatrix} -1 \cdot R & 5 \cdot G & -1 \cdot B \end{bmatrix}_{22} & \begin{bmatrix} 0 & -1 \cdot G & 0 \end{bmatrix}_{23} \\ \begin{bmatrix} 0 & -1 \cdot G & 0 \end{bmatrix}_{31} & \begin{bmatrix} -1 \cdot R & 5 \cdot G & -1 \cdot B \end{bmatrix}_{32} & \begin{bmatrix} 0 & -1 \cdot G & 0 \end{bmatrix}_{33} \end{bmatrix},$$

this is because our kernels are missing a *dimension* and currently not quite compatible with the

image array. Ergo, new kernels are derived by applying the concept of **broadcasting**.

$$Sharpen = \begin{bmatrix} \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} & \begin{bmatrix} -1 \\ 5 \\ -1 \end{bmatrix} & \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} \end{bmatrix}, Blur_{Gaussian3x3} = \begin{bmatrix} \begin{bmatrix} \frac{1}{16} \\ \frac{1}{8} \\ \frac{1}{16} \end{bmatrix} & \begin{bmatrix} \frac{1}{8} \\ \frac{1}{4} \\ \frac{1}{8} \end{bmatrix} & \begin{bmatrix} \frac{1}{16} \\ \frac{1}{8} \\ \frac{1}{16} \end{bmatrix} \end{bmatrix}$$

After obtaining the product from Eq. 8, we then want to add up all values in each channel to set up a new pixel $[R_{total} \ G_{total} \ B_{total}]$. With `np.sum(.)`, we can specify `axis=0` to sum by row first, or `axis=1` for column. Each `axis` will expectedly yield a different array, but both should share the same `(3, 3)` shape, which represents an array of 3 pixels. Then, we just need to sum these pixels with respect to their color components by stating `axis=0`.

2.4 6, 7 - Center and Circle cropping

2.4.1 Ideas

Center cropping is the most comfortable algorithm to implement since it does not require any extensive knowledge or formula. Finding the center of the image and attaining the dimension of the crop are all the pieces of the puzzle. Inputting a *factor* should be allowed so we can say how much in percentage of the image is retained.

Circle cropping can be a tad bit complicated if we haven't gotten a good grasp of the Cartesian coordinate system already. The general plan is to gauge how far a pixel is from the center of the crop, i.e., the circle, if this distance is greater than the radius, said pixel won't be in the output image. Same as **Center** cropping, we will also permit a *factor* parameter.

2.4.2 Implementations

Center cropping will be glossed over as the code of the `center_crop(.)` function is quite elementary to produce. Its core does not even require *NumPy* so the only thing worth writing about is the application of **slicing** to target the specific region belonging to the output image.

Circle cropping is the heart of this section as how it is implemented plays an integral role in Section 2.5 as well. To reiterate, for each pixel in the image, we want to acquire a `True/False` value denoting whether or not the coordinates of said pixel are within the radius. Figuring out which pixels lie inside the circle requires the knowledge of their indices. Rather than utilizing `np.arrange(.)` to generate an array of only indices, we involve a highly obscure but very specialized "*function*" called `np.ogrid`.

- The short article titled The Little Known OGrid Function in Numpy by Kharkar (2019) is a great introduction to `np.ogrid`.

After keeping the indices, or coordinates, of every pixel, to tell if it's in the circle, we can simply apply the **Circle's equation**, which essentially gives the **Euclidean distance** from the center to every pixel's location. Then, with *NumPy*'s **masking** concept, a pixel will be marked as *included* in the boolean mask if its distance to the center is less than the radius, otherwise we will *exclude* it. Lastly, note that the final mask is a matrix of booleans, `True` if the location is in the circle's range and `False` if not. Hence, this mask is the indicator for our desired region of pixels, we can take its inverse with the **tilde** $\sim$ operator in *NumPy* to select the pixels outside that region and color them to black (alkasm, 2020).

2.5 Extra - Cross Ellipses cropping

- *Cross Ellipses cropping* is not an officially recognized name in any document. Based solely on the characteristics of the shape, I impulsively coined the term.

2.5.1 Ideas

The strategy for **Cross Ellipses** cropping adheres to practically the same logic as **Circle** cropping in Section 2.4.1 apart from replacing the circle with an ellipse. Suppose there was only one ellipse and it was not skewed, we would estimate how far a pixel was from the center of the crop, if this distance put the pixel outside the span of the ellipse, it wouldn't be part of the output image. Yet, such was not the case for this **Extra** task since we are examining 2 symmetrically skewed ellipses. After proper consideration, solving said conundrum is not as daunting as it first seemed. We proceed with rotating one ellipse to the desired angle, then incorporating the earlier supposition's method, we have virtually procured all the pixels residing in our skewed ellipse. Ultimately, we create a mirror of said ellipse and appoint its pixels to also be ingredients of the resulting image.

2.5.2 Implementations

Fundamentally, to correctly skew the ellipse, we need to compute the angle, which can be done with a little bit of basic geometry. Although it is very straightforward to manually figure out, in

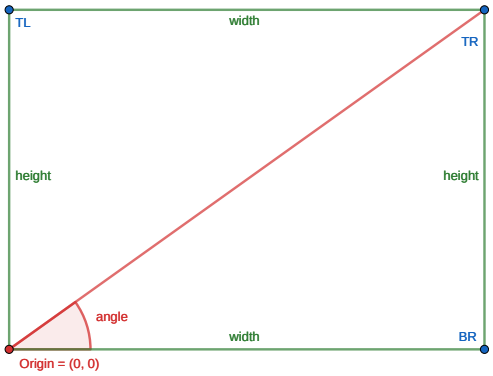

Figure 1. An oversimplification of any image. The green lines depict the skeleton of the image. The blue letters stand for (left to right): Top Left, Top Right, Bottom Right. The red line divides the image into halves, the bottom half triangle is the mean to determine the requisite angle.

the spirit of *NumPy*, we will use `np.arctan2(height, width)`. Having the angle, the next element we seek is an ellipse equation, the standard one fails to satisfy as it does not account for a rotation of any kind. A quick search with the keywords "rotated ellipse equation" and we can find an incredibly elegant equation put forward by user andikat dennis (2020) on Mathematics Stack Exchange.

$$\left(\frac{(x-h)\cos(A)+(y-k)\sin(A)}{a}\right)^2+\left(\frac{(x-h)\sin(A)-(y-k)\cos(A)}{b}\right)^2=1 \tag{9}$$

, "where h, k and a, b are the shifts and semi-axis in the x and y directions respectively and A is the angle measured from x axis." To stick closer to my code, let us rename a to `rad_x` and b to `rad_y`, suggesting the radius expanding to the x and y directions accordingly.

Inspecting Eq. 9, we can get x and y from `np.ogrid` the same way as in Section 2.4.2, our h and k are the center's x and y, and A is the angle we've just measured. Crossing off

everything that are already known, `rad_x` and `rad_y` are the only things left unidentified. A naive approach would be to set `rad_x` to half the diagonal's length scaled down by a factor, let's say 0.8, and `rad_y` to half the image's height. As able to be observed from Fig. 2, such

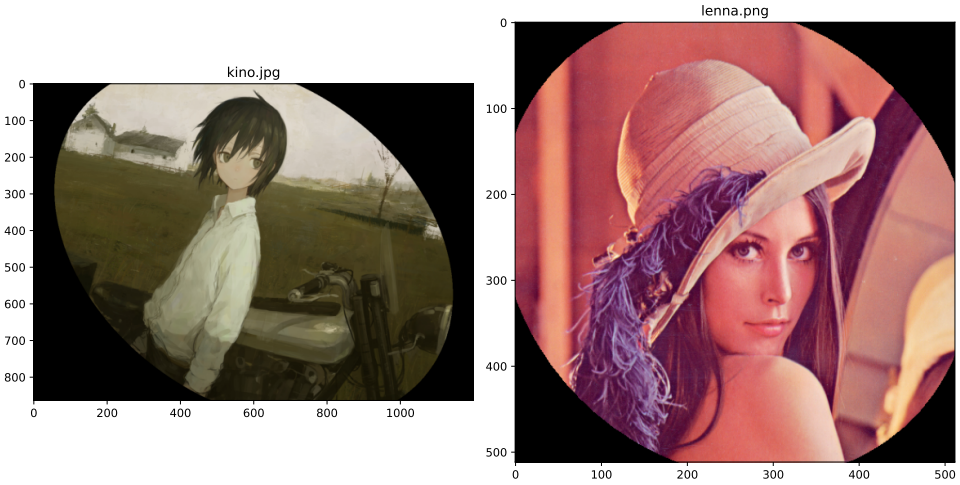

Figure 2. Results of setting `rad_x = diag_len / 2 * 0.8` and `rad_y = img_height / 2`. The ellipses can be seen clearly clipping outside the constraints of the images.

arbitrary assignment produces subpar outcomes as we neglected taking the area of the ellipse into account. One moderately better way that can only be realized through programming is, first fixing either `rad_x` or `rad_y`, then algorithmically try to find the other's value until the ellipse remains wholly within the image. This can be implemented with numerous popular searching algorithms, e.g., Binary Search.

Having said that, as we are in the mindset of *NumPy* and principally, **maths**, it would feel distasteful if we didn't try to decode it mathematically. That line of reasoning was what drive me to sink hours on end trying to work it out with a lot of concepts, most notably, circumscribed and inscribed rectangle, which I won't elaborate on since it would deviate from the main point. Eventually, I did give up and started scouring the web again, the keywords that time were "ellipse in rectangle". Ellipse in a Rectangle, that was a page I had clicked on and skipped over a good deal of times. In hindsight, many hours could have been saved if I had just bothered to merely scroll through it. Because lying at the bottom of that page, submitted by user Hypergeometricx (2017) is the wonderful equation serving precisely our craved purpose.

$$(\frac{x}{a})^2 + (\frac{y}{b})^2 - \frac{2xy}{ab}\sin(u-v) = \cos^2(u-v) \tag{10}$$

Notice that Eq. 10 are placing its center at the origin, we can alter this by switching all xs and ys to $(x-h)$s and $(y-k)$s as in Eq. 9, and $u-v$ is once again our angle.

$$(\frac{x-h}{a})^2 + (\frac{y-k}{b})^2 - \frac{2(x-h)(y-k)}{ab}\sin(A) = \cos^2(A) \tag{11}$$

Administering Eq. 11 right now would work fine with every image other than those with square dimensions, a discrepancy in the scaling of `rad_x` is the culprit. Luckily, this can be painlessly rectified with a simple check, if the image has a square shape, `rad_x` is set to half the image's width, otherwise we keep it as $diag\_len \div 2 \cdot 0.8$. The end result is an ideally rotated ellipse snuggling tightly within the limits of the image, as presented in Fig. 3.

Now to go about mirroring this ellipse, all we need to do is replacing the angle with its

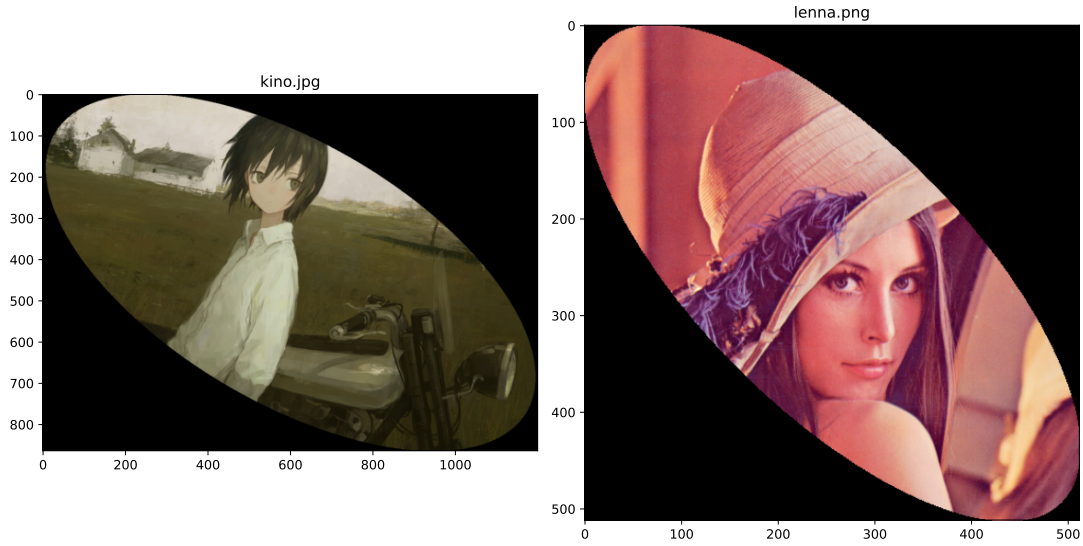

Figure 3. The ellipses accurately positioned.

negation in Eq. 11. Afterwards, with the same **masking** technique in 2.4.2, we will have ourselves 2 elliptical masks, one going from **TL** to **BR** and the other from **TR** to **Origin**, see Fig. 1. To merge these 2 masks into one giving us every pixel that would be in the output image, i.e., said pixel's location was labeled as `True`, `np.logical_or(.)` or the overloaded **pipe** `|` operator can be employed. And just like in Section 2.4.2, we take the inverse of this final mask with the **tilde** $\sim$ operator picking the pixels outside the 2 ellipses and dye them black.

Albeit rather insignificant, there is one finishing touch we should add to the mask. Keener eyes might have noticed that our ellipses, especially in the square image, are looking faintly slender, in Fig. 3. The swiftest way to mend this is by slightly changing the rotation angle. Off the back of bruteforcing some numbers while glancing at a **Unit circle** chart, I had discovered that rotating our angle clockwise by $\frac{5\pi}{12}$, i.e., $angle - \frac{5\pi}{12}$, granted the closest form to the example put up in the task's description of this project.

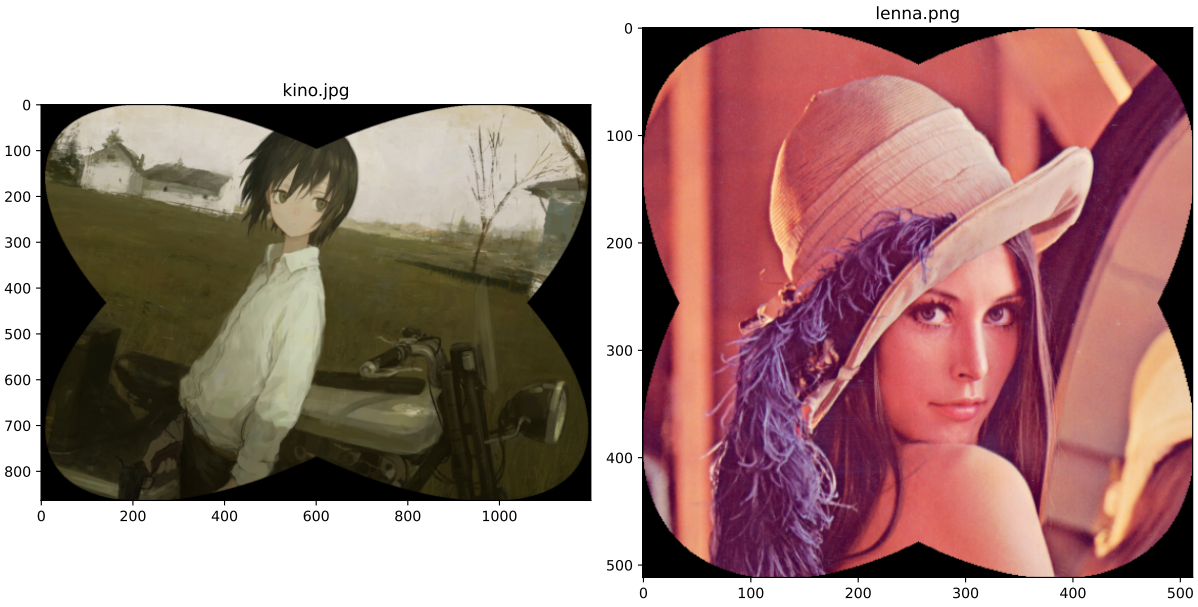

Figure 4. The images cropped by perfectly aligned cross ellipses.

3 RESULTS AND DISCUSSION

All samples used during this project were uploaded to my GitHub repository. For this section, only *kino.jpg* and *world_pulse.jpg* were selected. With *kino* being the primary exhibit and *world_pulse* is only used when I think the call for comparison arises.

3.1 1 - Brightness

The `factor` parameter was set to 0.5 and −0.5 correspondingly.

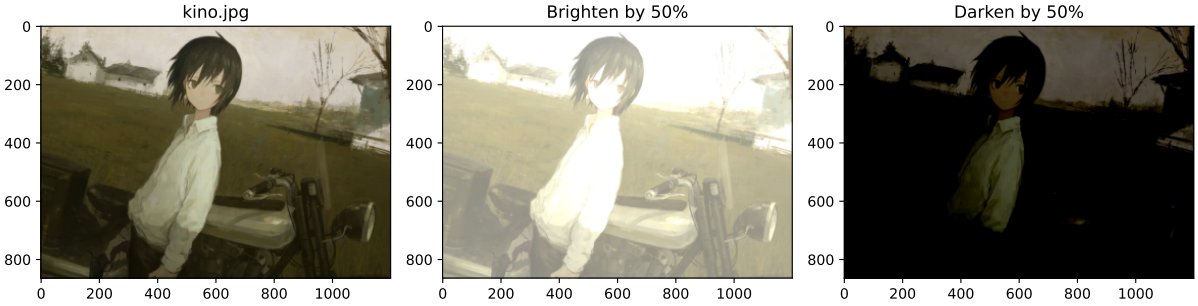

Figure 5. From left to right: The original image with a resolution of 1200 × 864 pixels, the image brightened by 50% and darkened by 50%. Runtime: 0.4s.

In Figure 5, *kino* has many pixels of near white color but the value of a channel is capped at 255, so when the brightening happened, those pixels turned thoroughly white. For darkening, the original image also maintains a somber tone, in consequence, the output's features became mostly indiscernible.

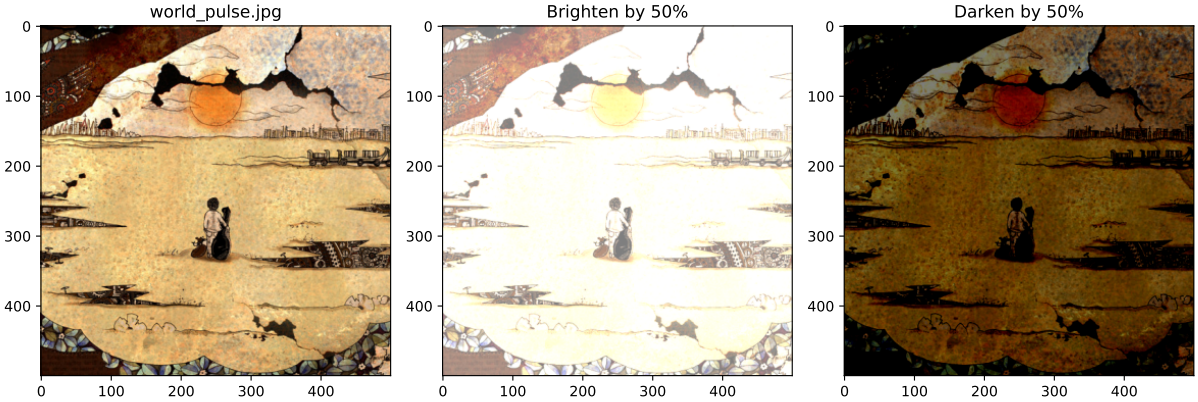

Figure 6. From left to right: The original image with a resolution of 500×500 pixels, the image brightened by 50% and darkened by 50%. Runtime: 0.3s.

Figure 6 presents a more vibrant image with a much smaller dimension. While it suffered the same fate when brightened, darkening supplied a better result. Incidentally, note that the runtime was somewhat improved, owing to the reduced size.

3.2 2 - Contrast

The `factor` parameter was set to 0.5 and −0.5 correspondingly.

Much like **Brightness** adjustment in Section 3.1, increasing contrast on a darker image, i.e., *kino* in Fig. 7, makes many of its aspects become difficult to recognize - a problem which was

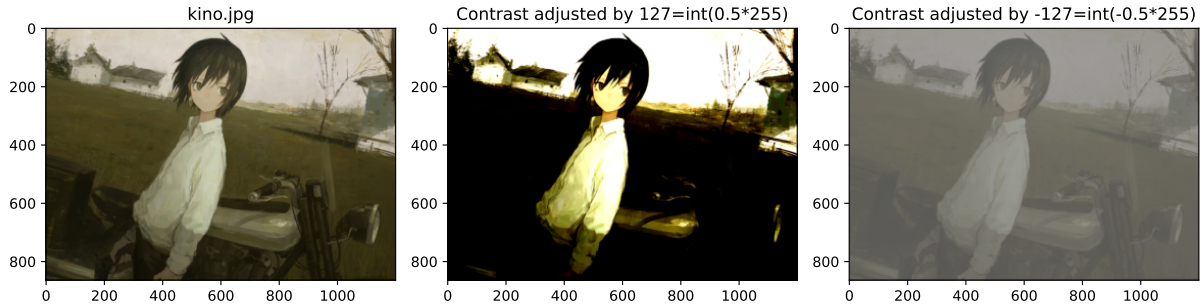

Figure 7. From left to right: The original image with a resolution of 1200×864 pixels, the image's contrast was modified by 127 and -127. Runtime: $0.5s$.

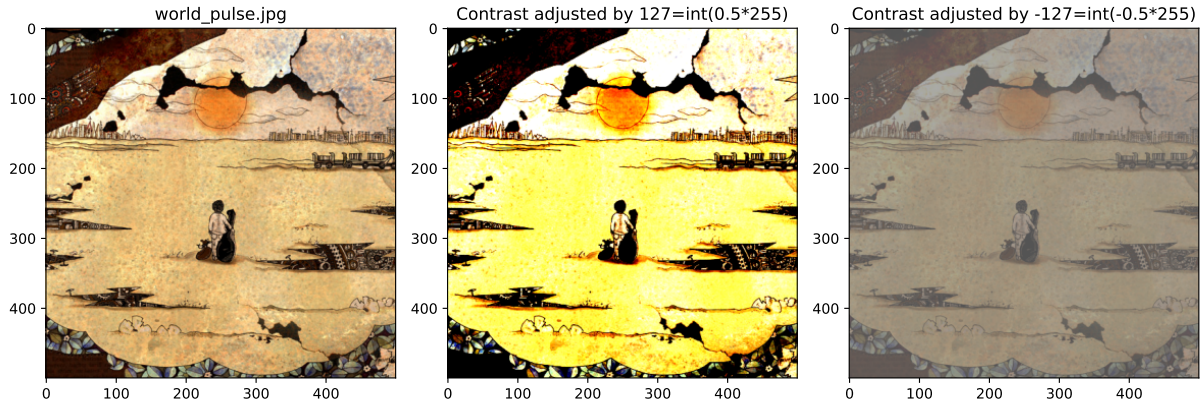

Figure 8. From left to right: The original image with a resolution of 500×500 pixels, the image's contrast was modified by 127 and -127. Runtime: $0.4s$.

not faced by *world_pulse* in Fig. 8. However, when contrast was removed from the base images, it only looked like a gray filter was donned on the images while hardly any details were lost.

3.3 3 - Flipping

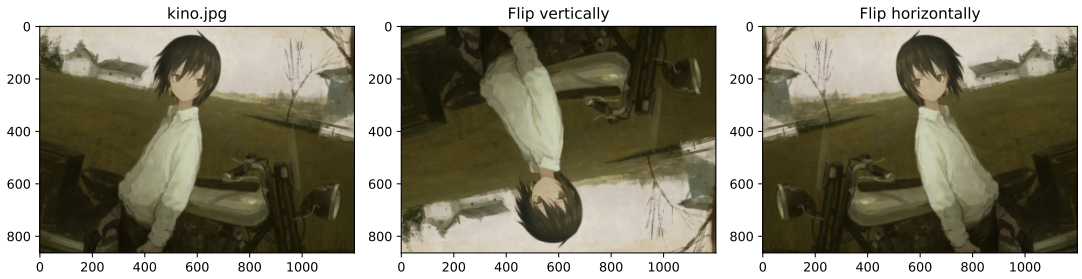

Figure 9. From left to right: The original image with a resolution of 1200×864 pixels, the image flipped vertically and horizontally. Runtime: $0.5s$.

With an acceptable runtime, the image was flipped both ways according to the requirement, as shown in Figure 9.

3.4 4 - Grayscale and Sepia

The `method` parameter was set to 0 for average and 1 for luminosity correspondingly.

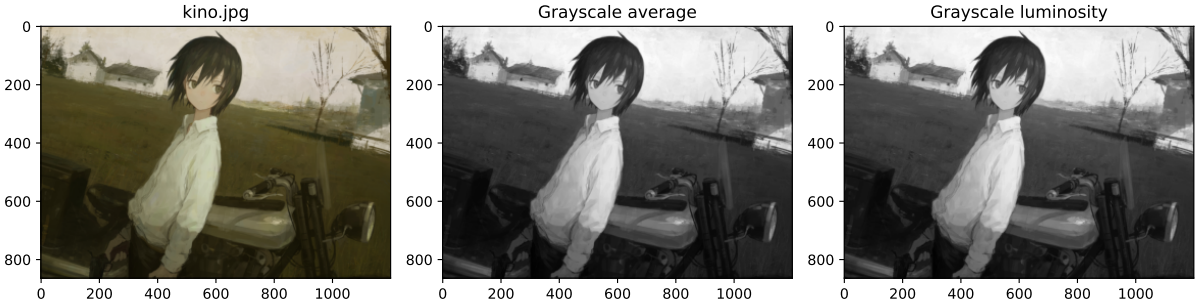

Figure 10. From left to right: The original image with a resolution of 1200×864 pixels, the image grayscaled with average and luminosity method. Runtime: $0.4s$.

Although the **Average** method remains conceptually inferior to **Luminosity**, with our naked eyes, the differences, if there had been any, were barely perceptible.

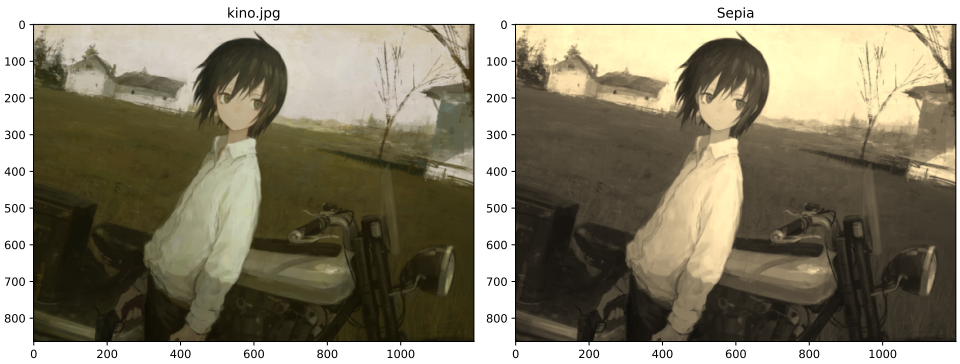

Figure 11. From left to right: The original image with a resolution of 1200×864 pixels, the image converted to Sepia. Runtime: $0.5s$.

Apart from the shifts in color, no other obvious changes to the image were detected. The runtime for one image to **Sepia** was greater than that of two to **grayscale** in Fig. 10. Reason being in the former, we had to compute the values with each individual channels before stacking them into the output, as opposed to the latter where we could just calculate the image as a whole.

3.5 5 - Blurring and Sharpening

Since both *RGB* and **grayscale** are supported, as described in Section 2.3.2, two versions of *kino* were used. Additionally, the difference between the original images and the results of doing a single blur could be quite tough to spot, that was why 10-time blurs were also dispensed.

Even with these large images, Fig. 12, **Blurring**'s runtime comfortably lay below the task's $15s$ requirement. Predictably, **grayscale** boasted impressively better runtimes as, in terms of color channels, the number of values was reduced and their datatype was less demanding to work with compared to *RGB*, i.e., the former had `int`s and the latter `np.ndarray`s.

Notice the blue lines showing up in the *RGB* image after it was **Sharpened**, Fig. 13, as we repeatedly put *kino* though the process, more of these lines would start appearing until eventually it became completely filled with noise. Due to that, the images were only sharpened once since subsequent outputs would have looked horrendous. Moreover, because **blurring** and **sharpening** shared the same `convolve(.)` function, the only distinction being the kernels, the runtimes stayed similar.

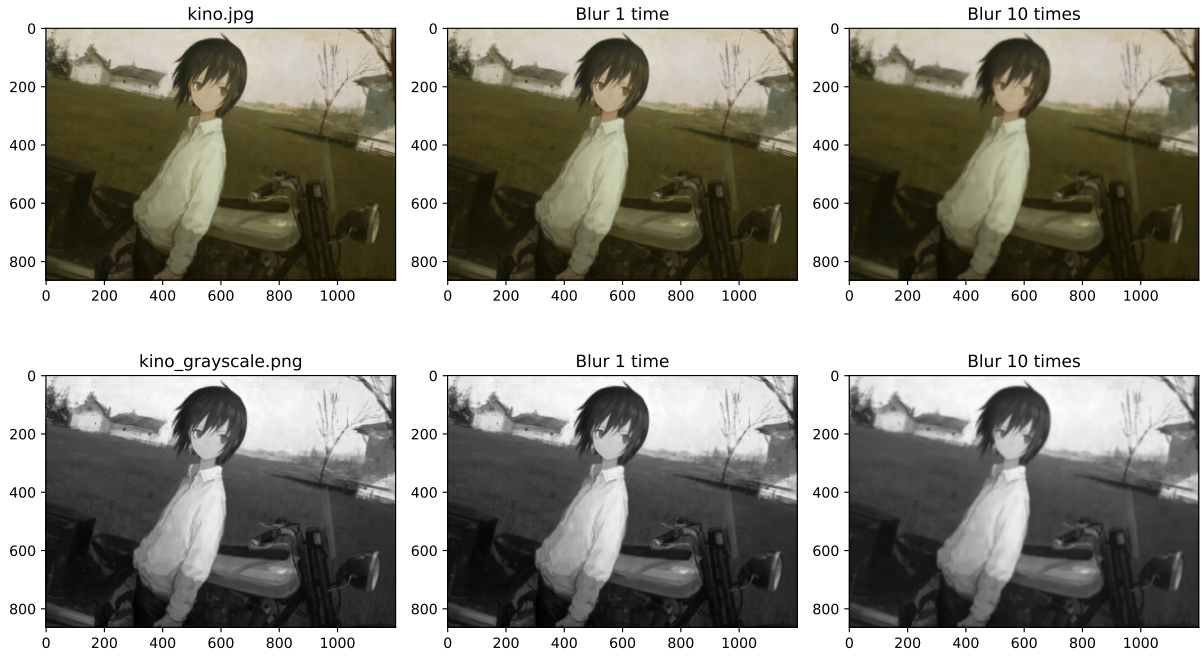

Figure 12. From left to right: The original images with resolutions of 1200×864 pixels, the images blurred once and 10 times. Runtime (left to right): 1^{st} row, $5.1s$ and $47s$; 2^{nd} row, $2.4s$ and $21.4s$.

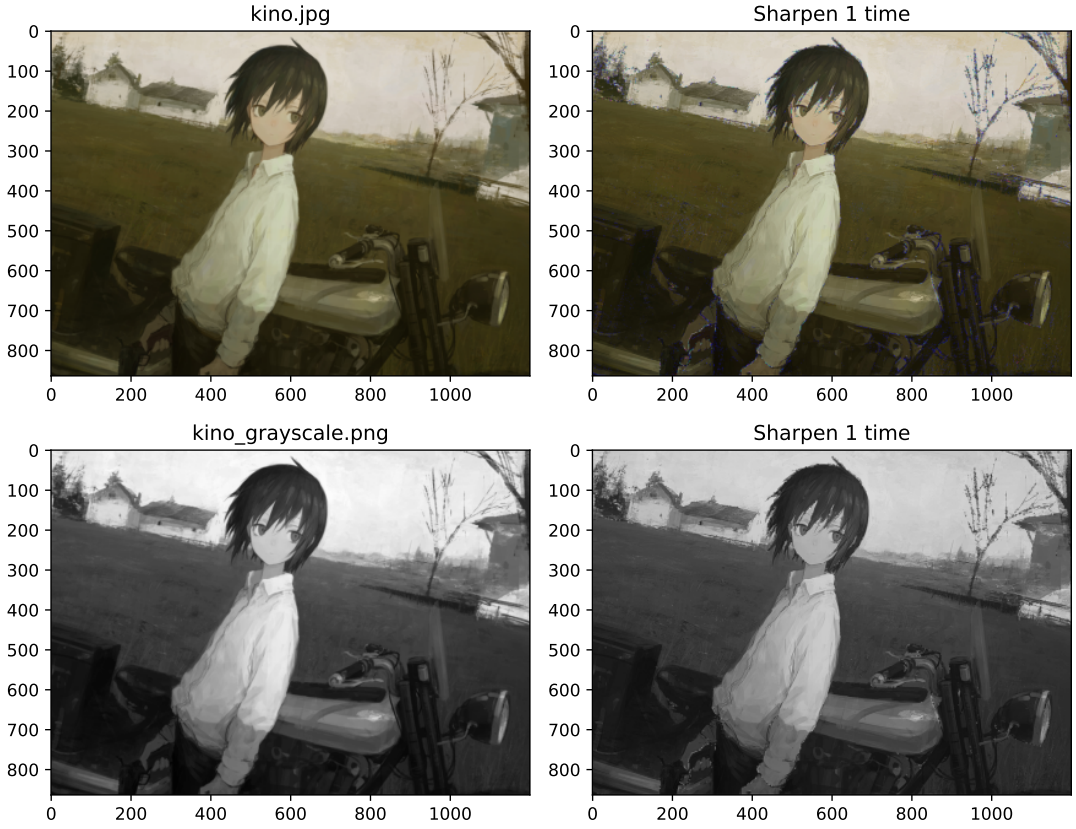

Figure 13. From left to right: The original images with resolutions of 1200×864 pixels, the images sharpened once. Runtime (top to bottom): $5.1s$ and $2.6s$.

3.6 6, 7 - Center and Circle cropping

Aside from `img:np.ndarray`, all other parameters of `center_crop(.)` and `circle_crop(.)` were left as defaults. On account of *kino* having the dimension of a rectangle, *world_pulse* was used to illustrate the outcomes a square image would produce.

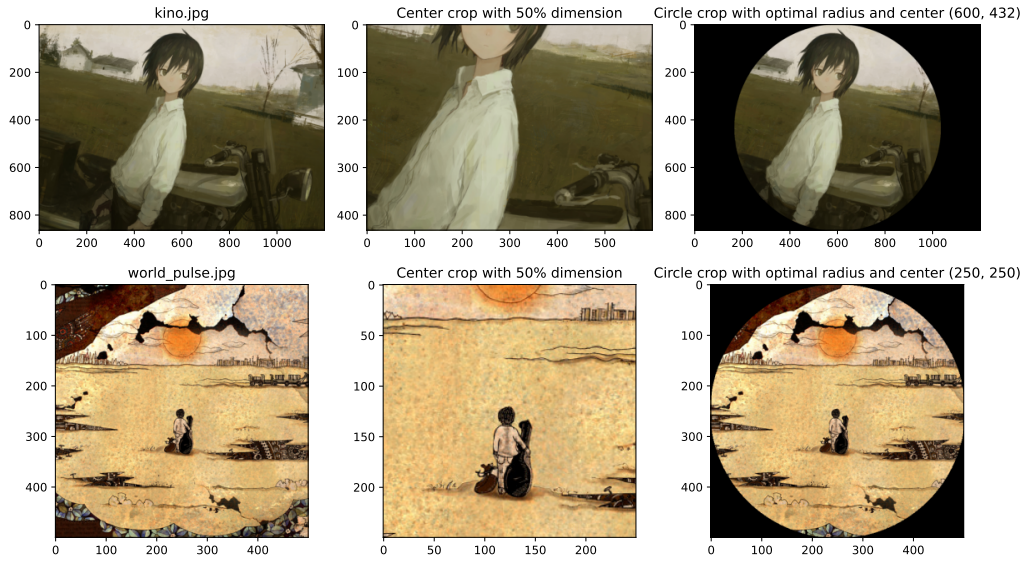

Figure 14. From left to right: 1st row, *kino* with a resolution of 1200×864 pixels, *kino* center and circle cropped; 2nd row, *world_pulse* with a resolution of 500×500 pixels, *world_pulse* center and circle cropped. Runtime (left to right): 1st row, $0.4s$ and $0.4s$; 2nd row, $0.4s$ and $0.3s$.

In the same manner as previous functions, in Fig. 14, the runtime rested on consistently low values, owing to the lack of iterations and computationally intensive algorithms, we mostly relied on maths and "*basic*" *Python*/*NumPy*'s operations.

3.7 Extra - Cross Ellipses cropping

Not unlike Section 3.6, save for `img:np.ndarray`, all other parameters of `cross_ellipses_crop(.)` were left as defaults, and *world_pulse* is brought to get a perspective on images of square shapes.

Figure 15. From left to right: The original image with a resolution of 1200×864 pixels, the image cross ellipses cropped Runtime: $0.4s$.

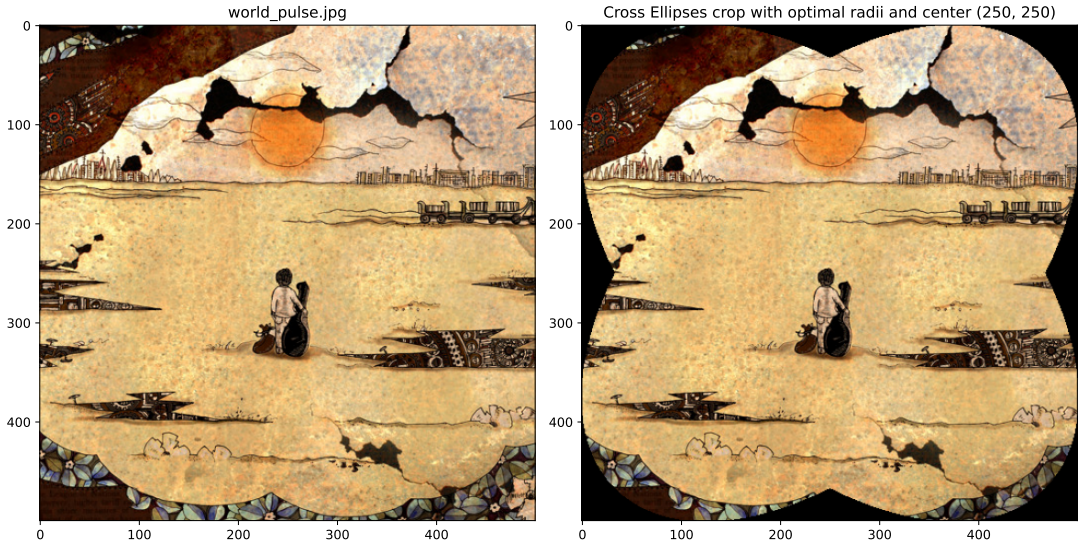

Figure 16. From left to right: The original image with a resolution of 500×500 pixels, the image cross ellipses cropped Runtime: $0.3s$.

Much of the points discussed in previous sections also held true for this one as well. With no remark left unsaid, it's only fitting to spend this time feasting our eyes upon these delightful end results of the **Cross Ellipses** cropping in Figs. 15 and 16.

SOFTWARE CITATIONS

This work uses the following software and packages:

- Python 3.11.4 (Van Rossum and Drake, 2009)
- NumPy 1.25.0 (Harris et al., 2020)
- Matplotlib 3.7.1 (Hunter, 2007)
- Pillow 10.0.0 (Clark, 2015)

DATA AVAILABILITY

All data and software used in this paper are public, their links are provided in the text when discussed. All data were used for educational and research purposes.

This project uses the following testing samples:

- *hcmus.jpg*: HCMUS by HCMUS (2023)
- *kino.jpg*: キノ by しおん (2017)
- *lenna.png*: Lenna by Forsén (1973)
- *world_pulse.jpg*: World Pulse - EP by 回路kairo (2013)

The structure and style of this report is based on:

- Abell 1201: Detection of an Ultramassive Black Hole in a Strong Gravitational Lens by Nightingale et al. (2023)

- ColourCompressionReport by Tong (2023) (TGHuybu)
- HCMUS-report-template by Tran (2023)
- Monthly Notices of the Royal Astronomical Society (MNRAS) LaTeX template and guide for authors by Press (2023)
- Simulating supermassive black hole mass measurements for a sample of ultra massive galaxies using ELT/HARMONI high spatial resolution integral-field stellar kinematics by Nguyen et al. (2023)
- template_HCMUT by Bình (2017)

ACKNOWLEDGEMENTS

I would like thank my lecturers of the current Applied Mathematics and Statistics course at Ho Chi Minh City University of Science (Vu Quoc Hoang, Nguyen Van Quang Huy, Le Thanh Tung, Phan Thi Phuong Uyen) for their guidance and support during theory classes and lab sessions, and mainly for this amazing project. In many ways, my time with **Project 02: Image Processing** was much more fulfilling and enjoyable compared to **Project 01: Color Compression**. I actually learned countless new things through this project and every line of code I wrote truly felt earned, as opposed to the latter where it was just a single *lackluster* algorithm.

As a final note, I want extend my deepest gratitude to Lecturer Phuong Uyen for suggesting the idea of writing a report in LaTeX, and especially my great astronomy-obsessed friend Huy G. Tong for inspiring me to actually go through with it, after seeing his nicely-typed document for the **Color Compression** project. This LaTeXreport would not have turned out as full-fledged as it now is if hadn't been for his **Project 01** report, the **MNRAS** template and all the black hole research papers he sent for me to use as references.

REFERENCES

alkasm. How can i create a circular mask for a numpy array? Stack Overflow, 2020. URL https://stackoverflow.com/a/44874588. (version: 2020-02-07).

andikat dennis. What is the general equation of the ellipse that is not in the origin and rotated by an angle? Mathematics Stack Exchange, 2020. URL https://math.stackexchange.com/q/434482. URL:https://math.stackexchange.com/q/434482 (version: 2020-12-13).

P. Antoniadis. How to convert an rgb image to a grayscale. *Baeldung on CS*, 2023. URL https://www.baeldung.com/cs/convert-rgb-to-grayscale.

C. T. Bình. template_hcmut. Overleaf, 2017. URL https://www.overleaf.com/latex/templates/template-hcmut/vwfqjwvrhfxq.

A. Clark. Pillow (pil fork) documentation, 2015. URL https://pillow.readthedocs.io/en/stable/?badge=latest.

denis. Combine 3 separate numpy arrays to an rgb image in python. Stack Overflow, 2012. URL https://stackoverflow.com/a/10463090. (version: 2012-05-05).

L. Forsén. Lenna, 1973. URL https://en.wikipedia.org/wiki/Lenna.

C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

HCMUS. Hcmus. hcmus, 2023. URL https://www.hcmus.edu.vn/component/sppagebuilder/?view=page&id=3151.

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9 (3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

Hypergeometricx. Ellipse in a rectangle. Mathematics Stack Exchange, 2017. URL https://math.stackexchange.com/q/2308342. URL:https://math.stackexchange.com/q/2308342 (version: 2017-06-03).

R. Kharkar. The little known ogrid function in numpy. *Towards Data Science*, 2019. URL https://towardsdatascience.com/the-little-known-ogrid-function-in-numpy-19ead3bdae40.

F. G. Loch. Image processing algorithms part 4: Brightness adjustment. *Dreamland Fantasy Studios*, 2010. URL https://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-4-brightness-adjustment/.

F. G. Loch. Image processing algorithms part 5: Contrast adjustment. *Dreamland Fantasy Studios*, 2015. URL https://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/.

D. D. Nguyen, M. Cappellari, and M. Pereira-Santaella. Simulating supermassive black hole mass measurements for a sample of ultra massive galaxies using elt/harmoni high spatial resolution integral-field stellar kinematics, 2023.

J. W. Nightingale, R. J. Smith, Q. He, C. M. O'Riordan, J. A. Kegerreis, A. Amvrosiadis, A. C. Edge, A. Etherington, R. G. Hayes, A. Kelly, J. R. Lucey, and R. J. Massey. Abell 1201: detection of an ultramassive black hole in a strong gravitational lens. *Monthly Notices of the Royal Astronomical Society*, 521(3):3298–3322, mar 2023. doi: 10.1093/mnras/stad587. URL https://doi.org/10.1093%2Fmnras%2Fstad587.

OpenCV. *The OpenCV Reference Manual*. OpenCV, 3.4 edition, July 2023. URL https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html.

R. K. P. How to get element-wise matrix multiplication (hadamard product) in numpy? Stack Overflow, 2018. URL https://stackoverflow.com/a/40035266. (version: 2018-07-12).

Patrick. Algorithm to modify brightness for rgb image? Stack Overflow, 2014. URL https://stackoverflow.com/a/24022126. (version: 2014-06-03).

T. Point. Brightness and contrast. *Tutorials Point*, 2023. URL https://www.tutorialspoint.com/dip/brightness_and_contrast.htm.

pooya kalahroodi. Imageblurring. GitHub, 2023. URL https://github.com/pooyakalahroodi/ImageBlurring.

M. P. Pound. How blurs & filters work. Computerphile, 2015. URL https://youtu.be/C_zFhWdM4ic.

O. U. Press. Monthly notices of the royal astronomical society (mnras) latex template and guide for authors. Overleaf, 2023. URL https://www.overleaf.com/latex/templates/monthly-notices-of-the-royal-astronomical-society-mnras-latex-template-and-guide-kqnjzrwjwjth.

Y. Shakeel. How to convert a color image into sepia image. *dyclassroom*, 2014. URL https://dyclassroom.com/image-processing-project/how-to-convert-a-color-image-into-sepia-image.

tel. Does np.dot automatically transpose vectors? Stack Overflow, 2021. URL https://stackoverflow.com/a/54161169. (version: 2021-04-26).

H. G. Tong. Colourcompressionreport. Overleaf, 2023. URL https://www.overleaf.com/read/fsqgtknvrwcg.

H.-Q. Tran. Hcmus-report-template. Overleaf, 2023. URL https://www.overleaf.com/latex/templates/hcmus-report-template/zyrhmsxynwqs.

G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

Wikipedia contributors. Kernel (image processing) — Wikipedia, the free encyclopedia, 2023a. URL https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&oldid=1162034482. [Online; accessed 23-July-2023].

Wikipedia contributors. Digital image processing — Wikipedia, the free encyclopedia, 2023b. URL https://en.wikipedia.org/w/index.php?title=Digital_image_processing&oldid=1163023125. [Online; accessed 22-July-2023].

しおん. キノ. pixiv, 2017. URL https://www.pixiv.net/en/artworks/65515709.

回路kairo. World pulse - ep. SoundCloud, 2013. URL https://soundcloud.com/k_a_i_r_o/world-pulse-xfd.