ELEC564 – Spring 2023
Homework 1

Student's Name: Hsuan-You (Shaun) Lin / Net ID: hl116
Student ID: S01435165

Link to Colab notebook: https://colab.research.google.com/drive/1srq7oF7rJeea7N1kwd7-0n1Atvl1mu2Y?usp=sharing

1.0 Basic Image Operations

1.1 Combining Two Images

a. Read in two large (> 256 x 256) images, A and B into your Colab notebook (see sample Colab notebook that was shared with the class earlier).

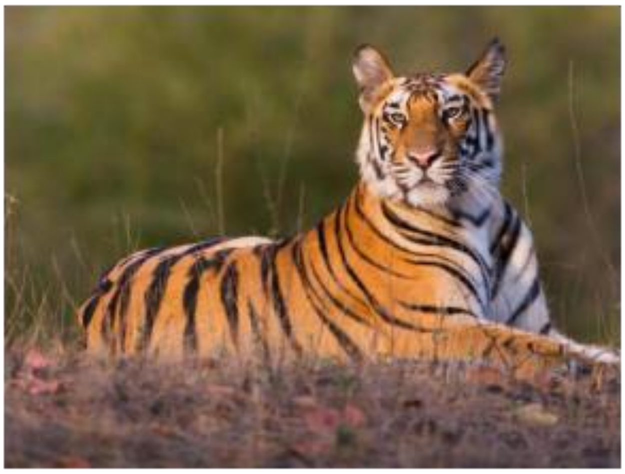


Figure 1. Input image A

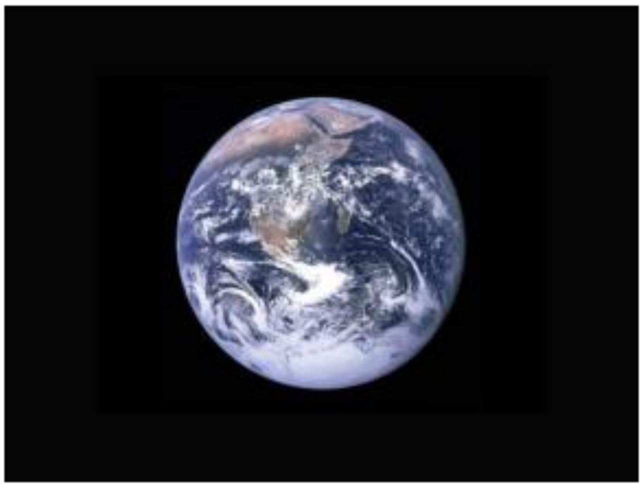


Figure 2. Input image B

b. Resize A to 256x256 and crop B at the center to 256x256.

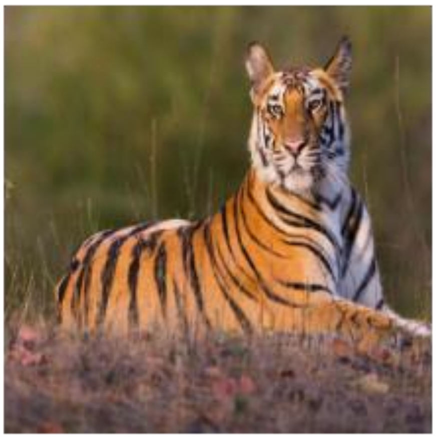


Figure 3. Output image A (resize to 256x256)

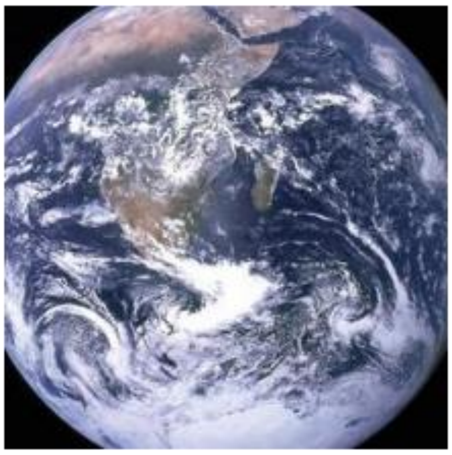


Figure 4. Output image B (resize at the center to 256x256)

c. Create a new image C such that the left half of C is the left half of A and the right half of C is the right half of B.

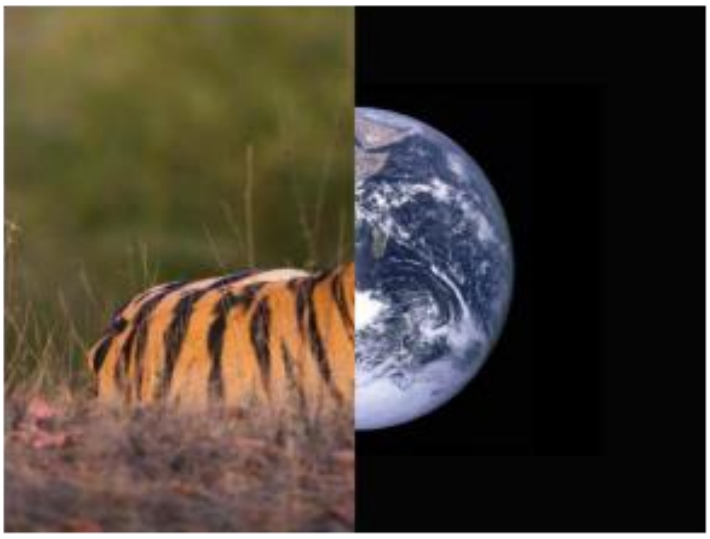


Figure 5. Output image C

d. Using a loop, create a new image D such that every odd numbered row is the corresponding row from A and every even row is the corresponding row from B.

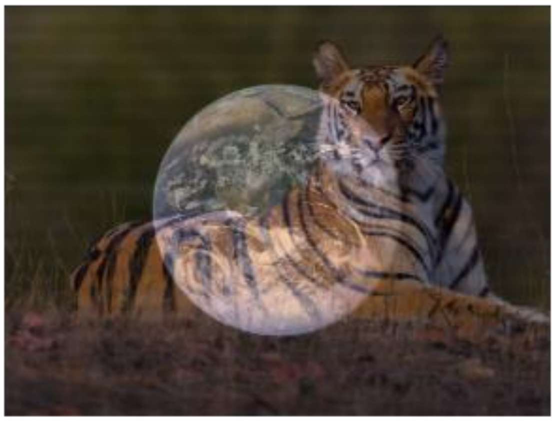
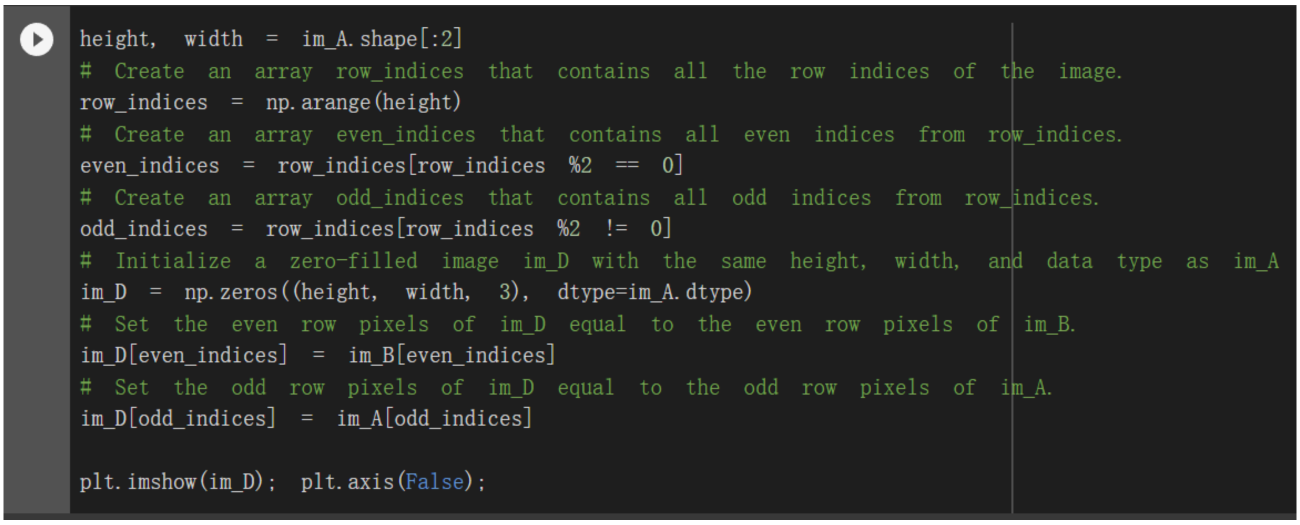


Figure 6. Output image D

e. Accomplish the same task in part d without using a loop. Describe your process.

My process:

I'm using NumPy's advanced indexing and broadcasting let even_indices arrays containing the indices of even rows, and another odd_indices arrays containing the indices of odd rows, simply using "%2" operator. This would be faster than using a loop, because it takes advantage of NumPy's vectorized operations.



```
height, width = im_A.shape[:2]
# Create an array row_indices that contains all the row indices of the image.
row_indices = np.arange(height)
# Create an array even_indices that contains all even indices from row_indices.
even_indices = row_indices[row_indices %2 == 0]
# Create an array odd_indices that contains all odd indices from row_indices.
odd_indices = row_indices[row_indices %2 != 0]
# Initialize a zero-filled image im_D with the same height, width, and data type as im_A
im_D = np.zeros((height, width, 3), dtype=im_A.dtype)
# Set the even row pixels of im_D equal to the even row pixels of im_B.
im_D[even_indices] = im_B[even_indices]
# Set the odd row pixels of im_D equal to the odd row pixels of im_A.
im_D[odd_indices] = im_A[odd_indices]

plt.imshow(im_D); plt.axis(False);
```

Figure 7. Python code without using a loop

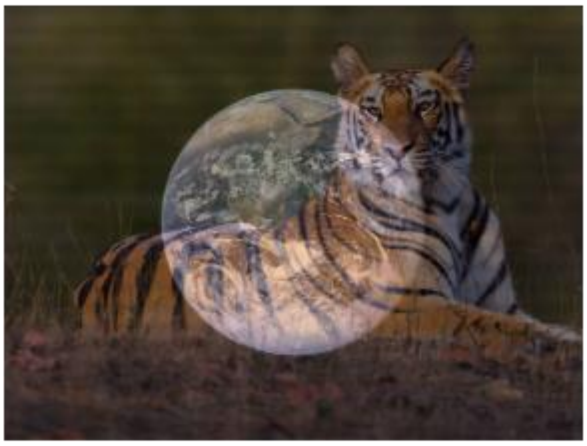


Figure 8. Output image D without using a loop

1.2 Color Spaces

a. Download the peppers image from this link. Return a binary image (only 0s and 1s), with 1s corresponding to only the yellow peppers. Do this by setting a minimum and maximum threshold value on pixel values in the R,G,B channels. Note that you won't be able to perfectly capture the yellow peppers, but give your best shot!

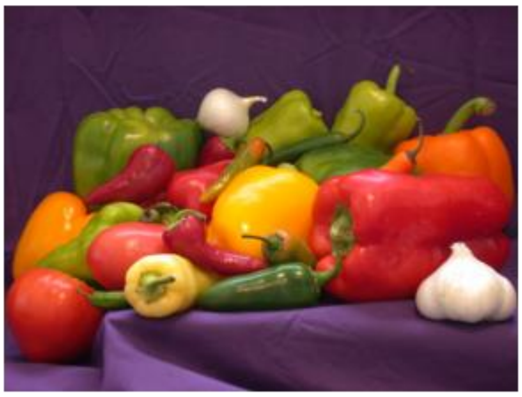


Figure 9. Input image

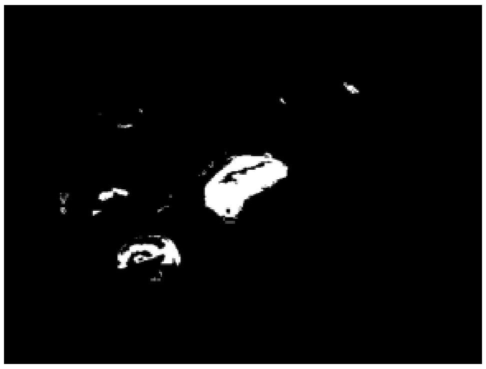


Figure 10. Output binary image with RGB color space

b. While RGB is the most common color space for images, it is not the only one. For example, one popular color space is HSV (Hue-Saturation-Value). Hue encodes color, value encodes lightness/darkness, and saturation encodes the intensity of the color. For a visual, see Fig. 1 of this wiki article. Convert the image to the HSV color space using OpenCV's cvtColor() function, and try to perform the same task by setting a threshold in the Hue channel.

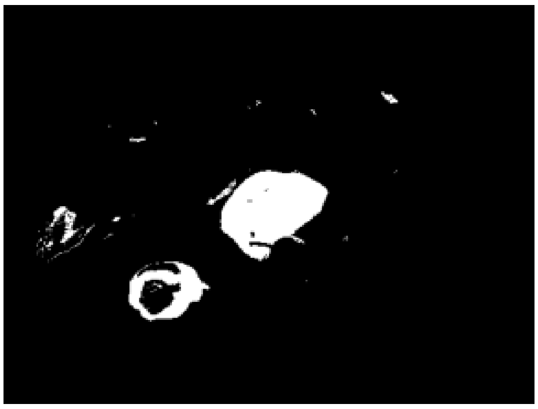


Figure 11. Output binary image with HSV color space

c. Add both binary images to your report. Which colorspace was easier to work with for this task, and why?

From my observations, HSV color space provides a more effective solution compared to the RGB color space. This is because the hue channel in the HSV color space specifically represents the color of an object, allowing for more precise color separation. In contrast, the RGB color space requires setting threshold values on individual R, G, and B channels, which can often result in incorrect or inconsistent separations.

In *Figure 10*, when using the RGB color space, the binary image obtained is less precise. However, as seen in *Figure 11*, the binary image obtained in the HSV color space clearly separates the yellow peppers with a high degree of accuracy, resulting in a clear, concise binary representation of the yellow peppers.

In conclusion, using the HSV color space is a more effective solution for separating yellow peppers in an image due to its ability to specifically represent color information in the hue channel, leading to a more accurate binary image representation.

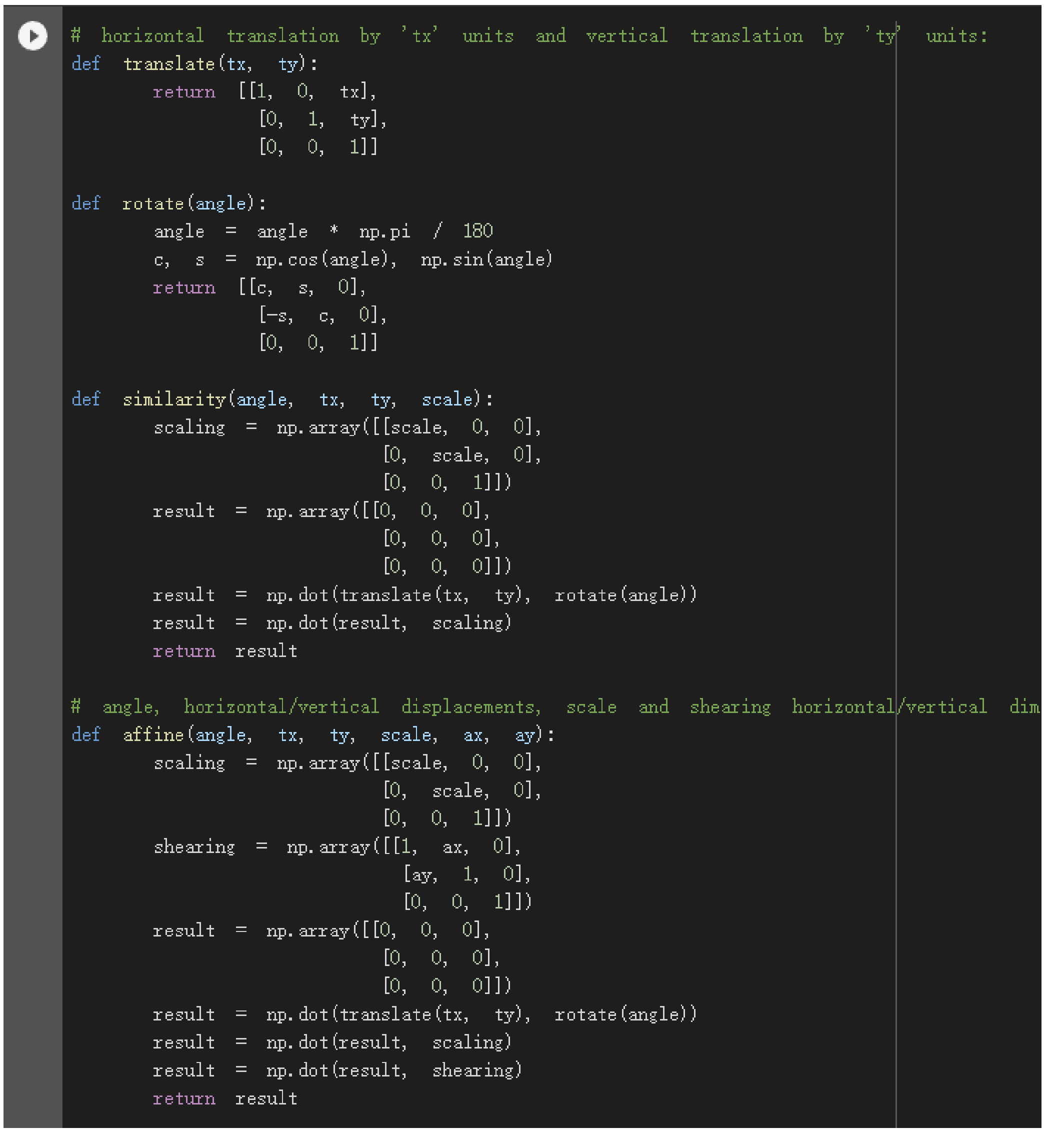
2.0 2D Geometric Transforms

2.1 Write functions to produce transformation matrices

Write separate functions that output the 3 x 3 transformation matrices for the following transforms: translation, rotation, similarity (translation, rotation, and scale), and affine. The functions should take as input the following arguments:

1. Translation: horizontal and vertical displacements
2. Rotation: angle
3. Similarity: angle, horizontal/vertical displacements, and scale factor (assume equal scaling for horizontal and vertical dimensions)
4. Affine: 6 parameters

The output of each function will be a 3 x 3 matrix



```
# horizontal translation by ’tx’ units and vertical translation by ’ty’ units:
def translate(tx, ty):
        return [[1, 0, tx],
                [0, 1, ty],
                [0, 0, 1]]

def rotate(angle):
        angle = angle * np.pi / 180
        c, s = np.cos(angle), np.sin(angle)
        return [[c, s, 0],
                [-s, c, 0],
                [0, 0, 1]]

def similarity(angle, tx, ty, scale):
        scaling = np.array([[scale, 0, 0],
                            [0, scale, 0],
                            [0, 0, 1]])
        result = np.array([[0, 0, 0],
                           [0, 0, 0],
                           [0, 0, 0]])
        result = np.dot(translate(tx, ty), rotate(angle))
        result = np.dot(result, scaling)
        return result

# angle, horizontal/vertical displacements, scale and shearing horizontal/vertical dim
def affine(angle, tx, ty, scale, ax, ay):
        scaling = np.array([[scale, 0, 0],
                            [0, scale, 0],
                            [0, 0, 1]])
        shearing = np.array([[1, ax, 0],
                             [ay, 1, 0],
                             [0, 0, 1]])
        result = np.array([[0, 0, 0],
                           [0, 0, 0],
                           [0, 0, 0]])
        result = np.dot(translate(tx, ty), rotate(angle))
        result = np.dot(result, scaling)
        result = np.dot(result, shearing)
        return result
```

Figure 12. 4 types transformation function Python code

2.2 Write a function that warps an image with a given transformation matrix

Next, write a function imwarp(I, T) that warps image I with transformation matrix T. The function should produce an output image of the same size as I. See Fig. 1 for an example of a warp induced by a rotation transformation matrix. Make the origin of the coordinate system correspond to the CENTER of the image, not the top-left corner. This will result in more intuitive results, such as how the image is rotated around its center in Fig. 1.

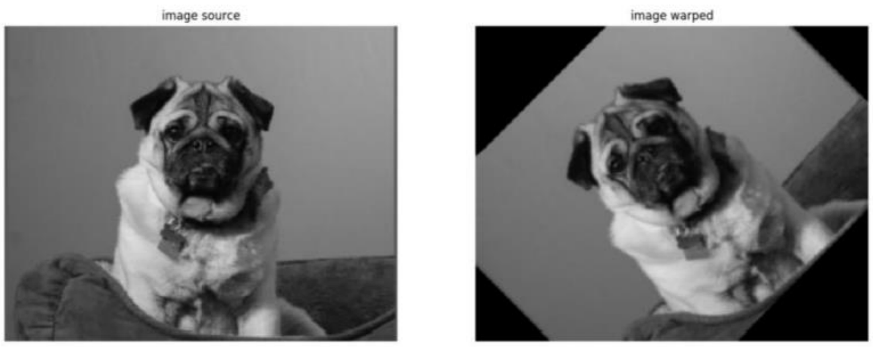


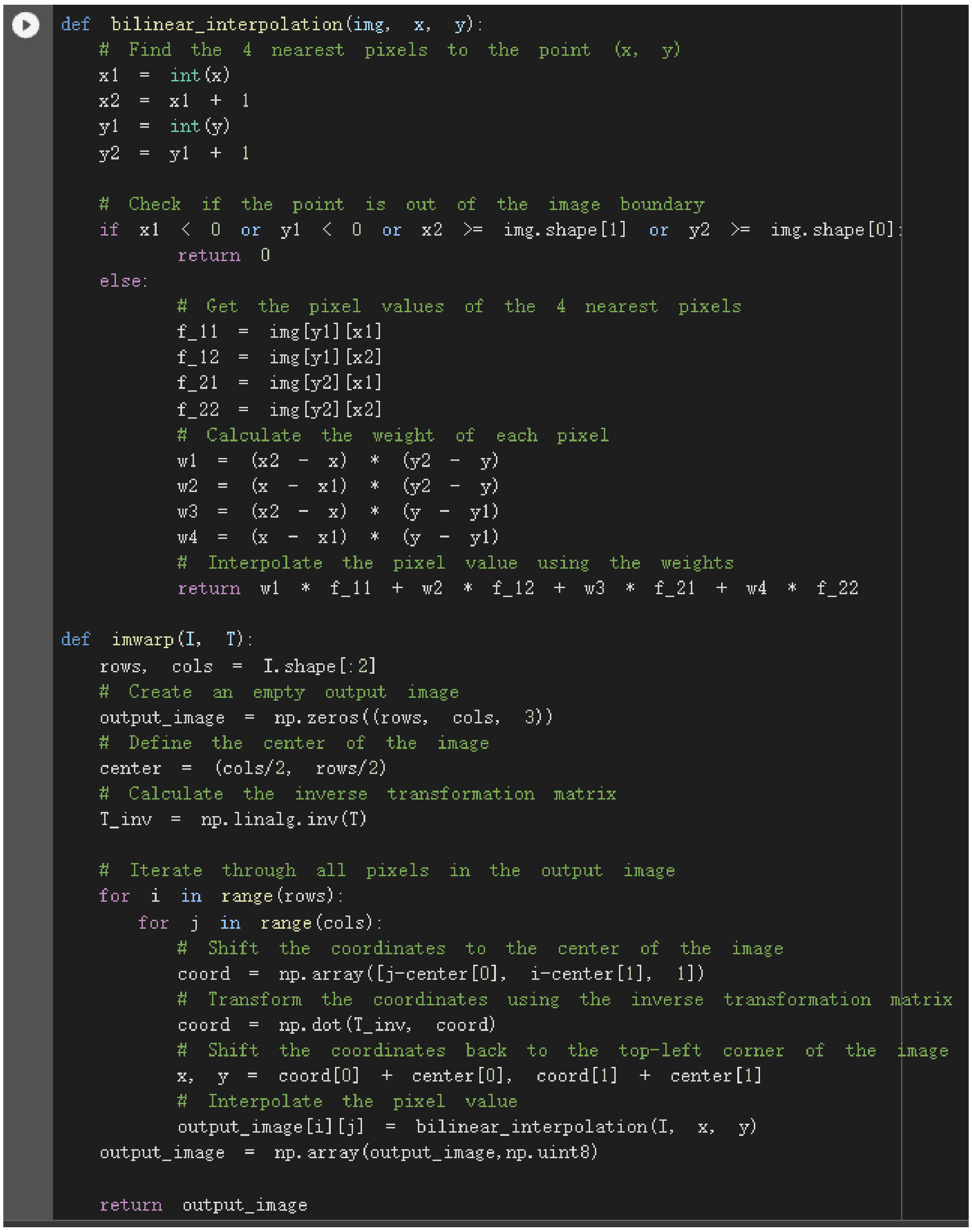
Fig. 1: Example of an input image (left) transformed by a rotation matrix, resulting in a 'warped' image (right).

Hint 1: Consider the transformation matrix T to describe the mapping from each pixel in the output image back to the original image. By defining T in this way, you can account for each output pixel in the warp, resulting in no 'holes' in the output image (see Lec. 03 slides).

Hint 2: What happens when the transformation matrix maps an output pixel to a non-integer location in the input image? You will need to perform bilinear interpolation to handle this correctly (see Lec. 03 slides).

Hint 3: You may find NumPy's meshgrid function useful to generate all pixel coordinates at once, without a loop.

Figure 13. Input image



```
def bilinear_interpolation(img, x, y):
    # Find the 4 nearest pixels to the point (x, y)
    x1 = int(x)
    x2 = x1 + 1
    y1 = int(y)
    y2 = y1 + 1

    # Check if the point is out of the image boundary
    if x1 < 0 or y1 < 0 or x2 >= img.shape[1] or y2 >= img.shape[0]:
            return 0
    else:
            # Get the pixel values of the 4 nearest pixels
            f_11 = img[y1][x1]
            f_12 = img[y1][x2]
            f_21 = img[y2][x1]
            f_22 = img[y2][x2]
            # Calculate the weight of each pixel
            w1 = (x2 - x) * (y2 - y)
            w2 = (x - x1) * (y2 - y)
            w3 = (x2 - x) * (y - y1)
            w4 = (x - x1) * (y - y1)
            # Interpolate the pixel value using the weights
            return w1 * f_11 + w2 * f_12 + w3 * f_21 + w4 * f_22

def imwarp(I, T):
    rows, cols = I.shape[:2]
    # Create an empty output image
    output_image = np.zeros((rows, cols, 3))
    # Define the center of the image
    center = (cols/2, rows/2)
    # Calculate the inverse transformation matrix
    T_inv = np.linalg.inv(T)

    # Iterate through all pixels in the output image
    for i in range(rows):
        for j in range(cols):
            # Shift the coordinates to the center of the image
            coord = np.array([j-center[0], i-center[1], 1])
            # Transform the coordinates using the inverse transformation matrix
            coord = np.dot(T_inv, coord)
            # Shift the coordinates back to the top-left corner of the image
            x, y = coord[0] + center[0], coord[1] + center[1]
            # Interpolate the pixel value
            output_image[i][j] = bilinear_interpolation(I, x, y)
    output_image = np.array(output_image,np.uint8)

    return output_image
```

Figure 14. Bilinear interpolation function & imwarp function Python code

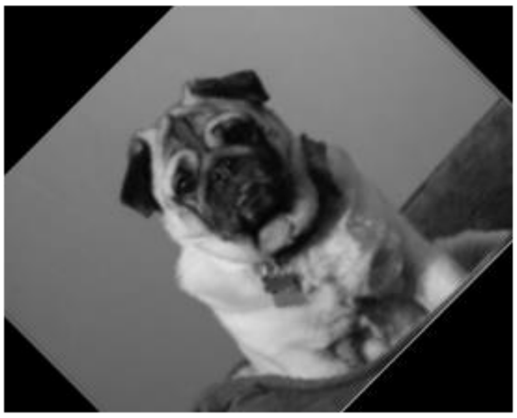
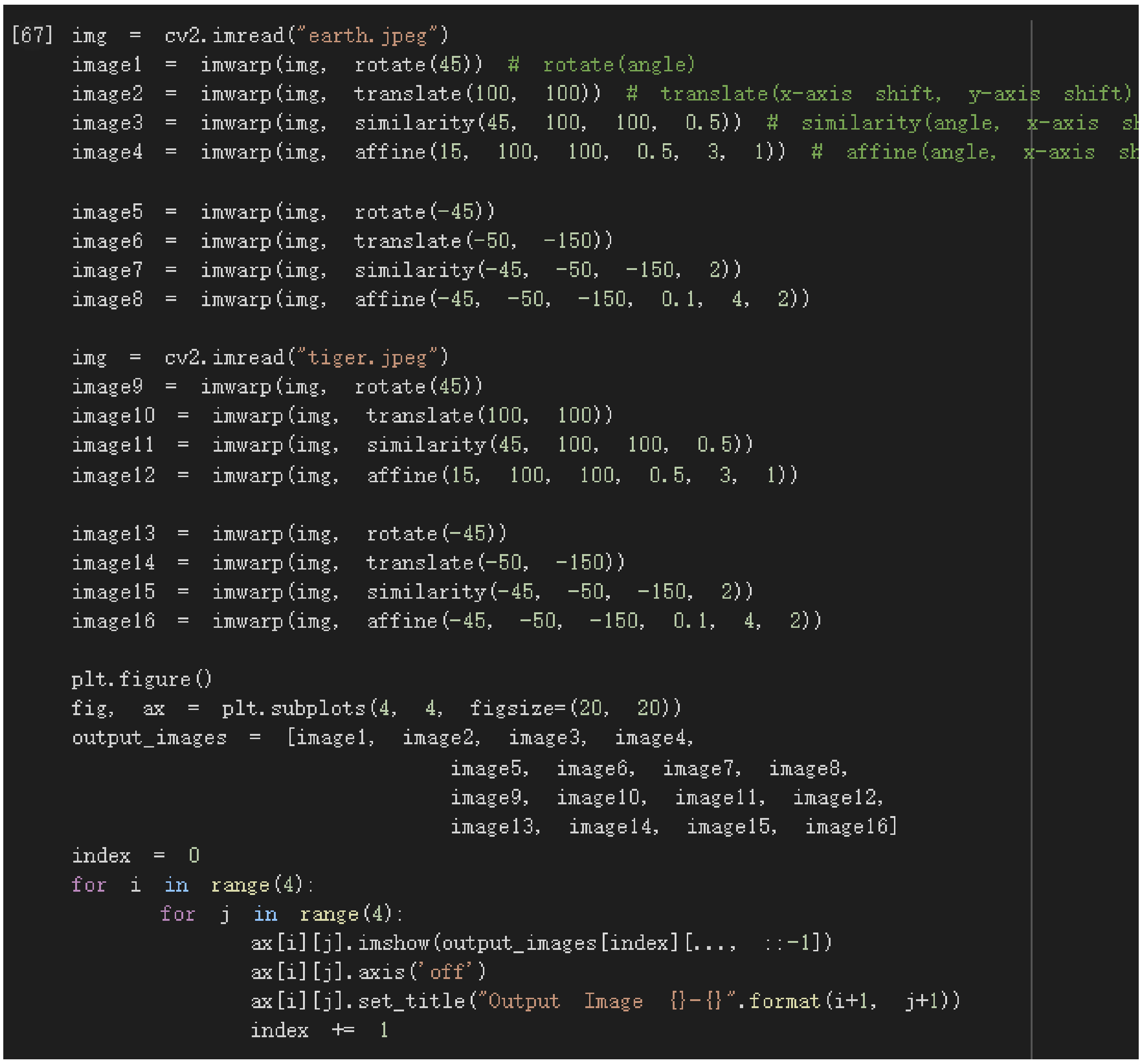


Figure 15. Output image using rotate transformation function

2.3 Demonstrate your warping code on two color images of your choice.

For each of the two images, show 2-3 transformations of each type (translation, rotation, similarity, affine) in your report.



```
[67] img = cv2.imread("earth.jpeg")
     image1 = imwarp(img, rotate(45)) # rotate(angle)
     image2 = imwarp(img, translate(100, 100)) # translate(x-axis shift, y-axis shift)
     image3 = imwarp(img, similarity(45, 100, 100, 0.5)) # similarity(angle, x-axis sh
     image4 = imwarp(img, affine(15, 100, 100, 0.5, 3, 1)) # affine(angle, x-axis sh

     image5 = imwarp(img, rotate(-45))
     image6 = imwarp(img, translate(-50, -150))
     image7 = imwarp(img, similarity(-45, -50, -150, 2))
     image8 = imwarp(img, affine(-45, -50, -150, 0.1, 4, 2))

     img = cv2.imread("tiger.jpeg")
     image9 = imwarp(img, rotate(45))
     image10 = imwarp(img, translate(100, 100))
     image11 = imwarp(img, similarity(45, 100, 100, 0.5))
     image12 = imwarp(img, affine(15, 100, 100, 0.5, 3, 1))

     image13 = imwarp(img, rotate(-45))
     image14 = imwarp(img, translate(-50, -150))
     image15 = imwarp(img, similarity(-45, -50, -150, 2))
     image16 = imwarp(img, affine(-45, -50, -150, 0.1, 4, 2))

     plt.figure()
     fig, ax = plt.subplots(4, 4, figsize=(20, 20))
     output_images = [image1, image2, image3, image4,
                                 image5, image6, image7, image8,
                                 image9, image10, image11, image12,
                                 image13, image14, image15, image16]
     index = 0
     for i in range(4):
            for j in range(4):
                   ax[i][j].imshow(output_images[index][..., ::-1])
                   ax[i][j].axis('off')
                   ax[i][j].set_title("Output Image {}-{}".format(i+1, j+1))
                   index += 1
```

Figure 16. Different transformation function Python code

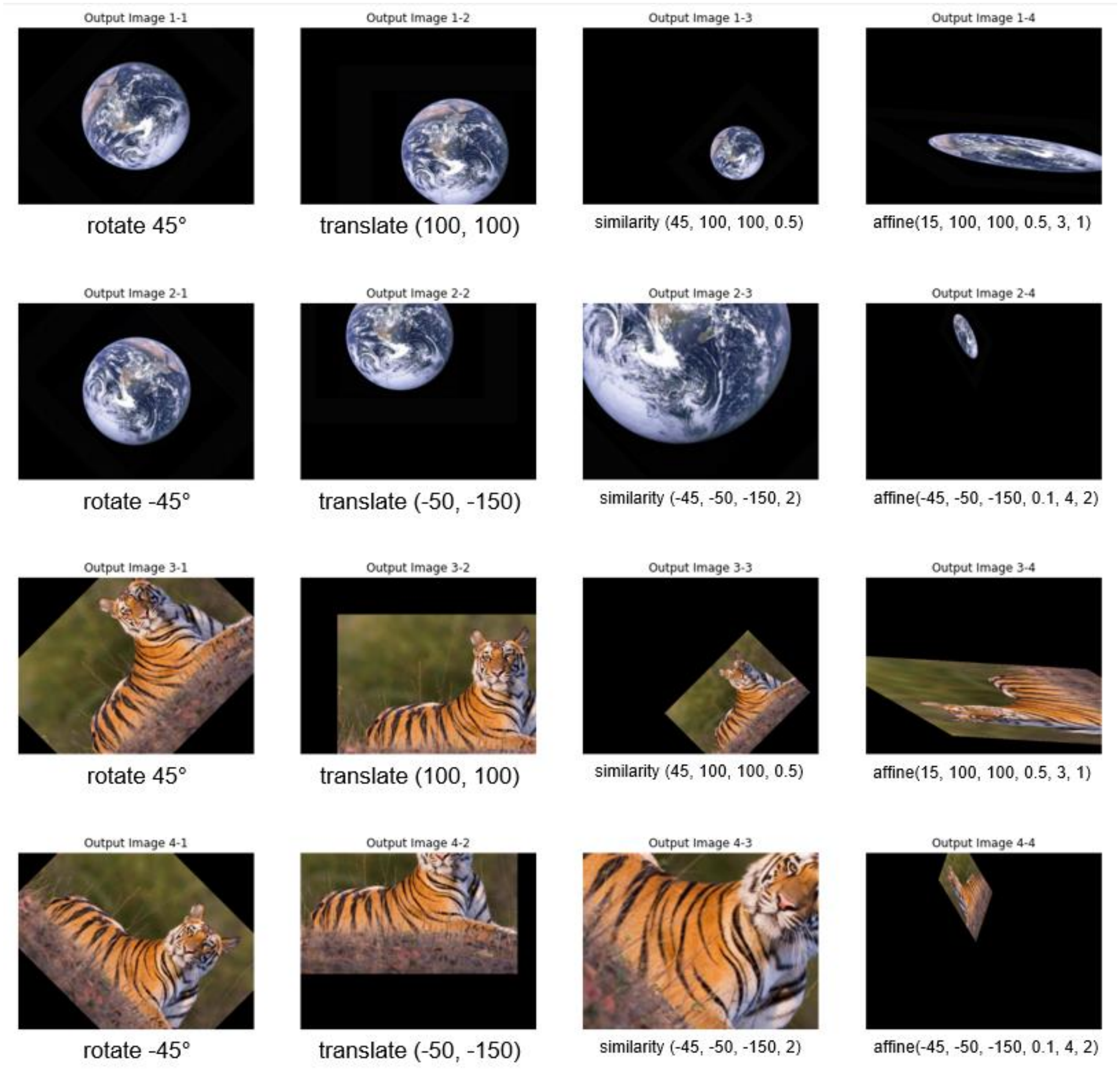


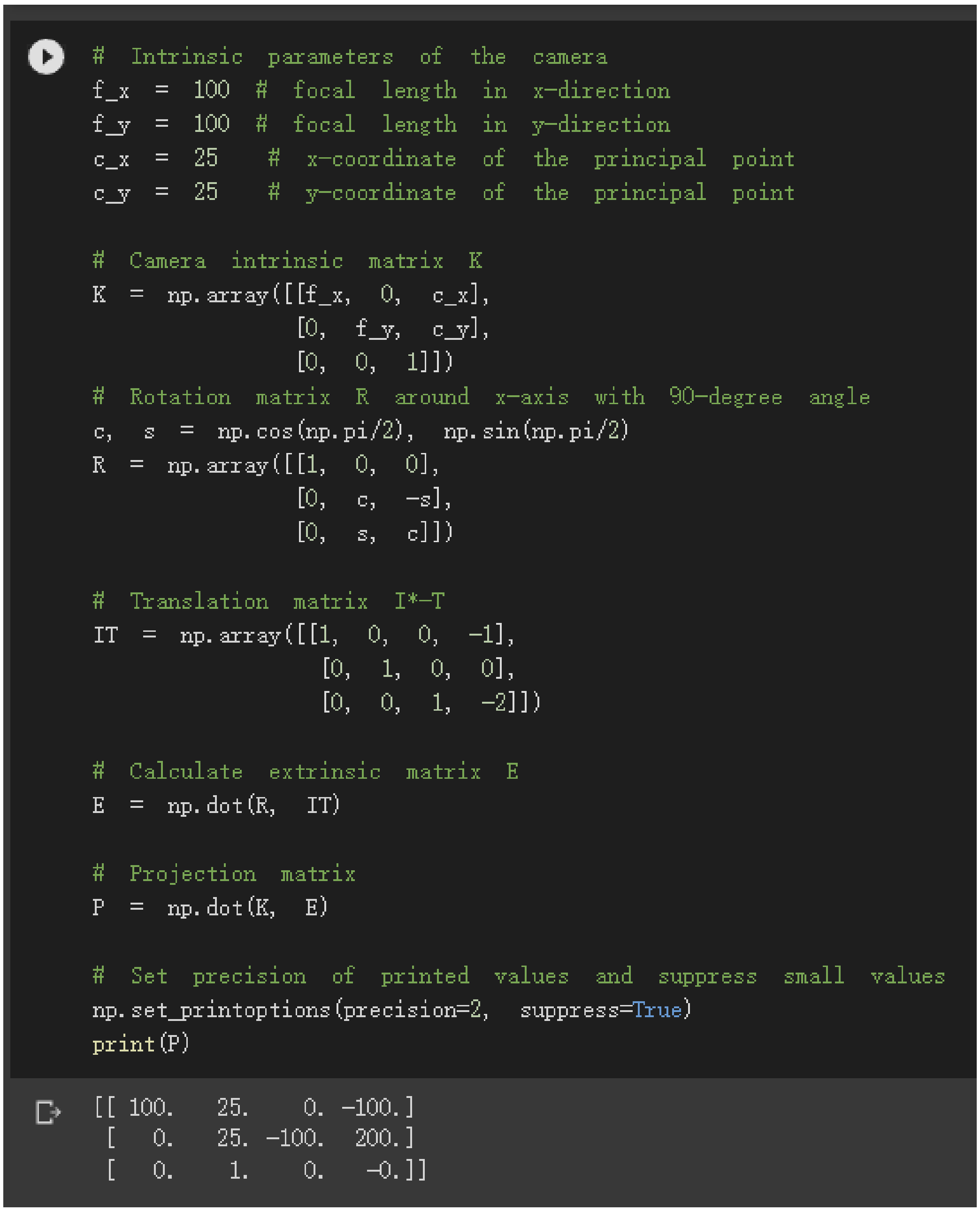
Figure 17. Output image using 2 different image, and 4 transformations and apply different parameters for each transformations

3.0 Cameras

3.1 Camera Matrix Computation

a. Calculate the camera intrinsic matrix K, extrinsic matrix E, and full rank 4 × 4 projection matrix P = KE for the following scenario with a pinhole camera:

- **The camera is rotated 90 degrees around the x-axis, and is located at (1, 0, 2)^T.**
- **The focal lengths fx, fy are 100.**
- **The principal point (cx, cy)^T is (25, 25).**



```
# Intrinsic parameters of the camera
f_x = 100 # focal length in x-direction
f_y = 100 # focal length in y-direction
c_x = 25   # x-coordinate of the principal point
c_y = 25   # y-coordinate of the principal point

# Camera intrinsic matrix K
K = np.array([[f_x, 0, c_x],
              [0, f_y, c_y],
              [0, 0, 1]])
# Rotation matrix R around x-axis with 90-degree angle
c, s = np.cos(np.pi/2), np.sin(np.pi/2)
R = np.array([[1, 0, 0],
              [0, c, -s],
              [0, s, c]])

# Translation matrix I*-T
IT = np.array([[1, 0, 0, -1],
               [0, 1, 0, 0],
               [0, 0, 1, -2]])

# Calculate extrinsic matrix E
E = np.dot(R, IT)

# Projection matrix
P = np.dot(K, E)

# Set precision of printed values and suppress small values
np.set_printoptions(precision=2, suppress=True)
print(P)
```

```
[[ 100.   25.    0. -100.]
 [   0.   25. -100.  200.]
 [   0.    1.    0.   -0.]]
```

Figure 18. Calculate projection matrix Python code

3.2 Field of view and focal length

You are given two cameras with the exact same sensor. The first camera has a wider field-of- view (FOV) than the second, with all other camera parameters being the same. Which camera has a shorter focal length and why?

My Answer:

The camera with the wider field-of-view (FOV) has a shorter focal length as it is designed to capture a larger portion of the scene in the frame. A wider FOV is achieved by using a shorter focal length lens, which allows for a greater angle of view, resulting in more of the scene being captured in the frame. In contrast, a longer focal length lens will have a narrower FOV and will zoom in on a smaller portion of the scene. This results in a more focused and magnified image but with a smaller overall field of view.

In summary, the relationship between FOV and focal length is inversely proportional, meaning that as the FOV increases, the focal length decreases, and vice versa. This is why the first camera with the wider FOV has a shorter focal length.

4.0 Relighting

a. Capture the image of the scene by turning on LAMP1 only (image I1). Now capture an image by turning on LAMP2 only (image I2). Finally, capture the image with both LAMP1 and LAMP2 on (image I12). Load and display these images into your Colab notebook.

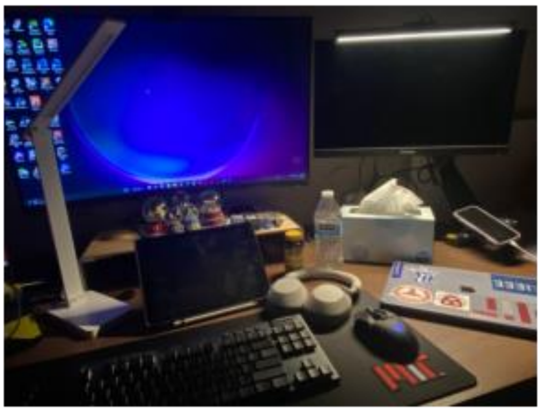


Figure 19. Input image I1

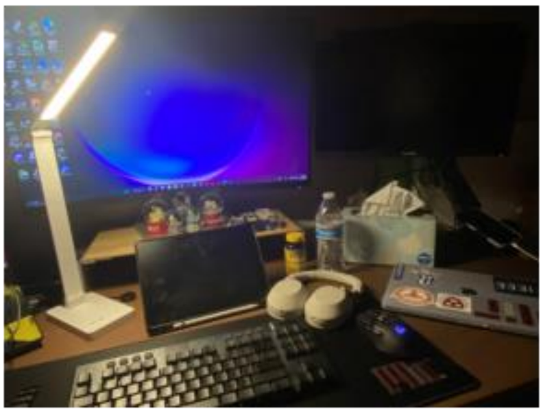


Figure 20. Input image I2

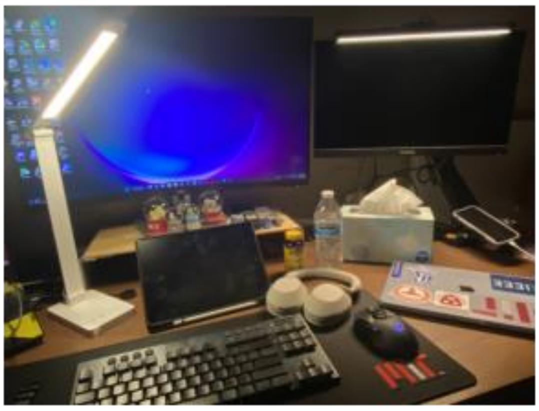


Figure 21. Input image I12

b. Now, you will create a synthetic photo (I12_synth) depicting the scene when both of the lamps are turned on by simply summing I1 and I2 together: I12_synth = I1 + I2. Also compute an image depicting the difference between the synthetic and real images: D = I12_synth - I12.

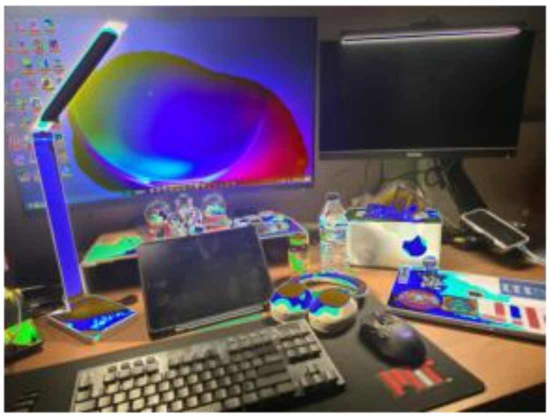


Figure 22. Output image I12_synth

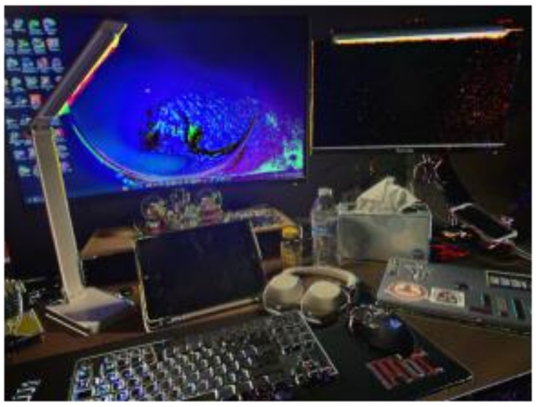


Figure 23. Output image D

c. In your report, show I1, I2, I12, I12_synth, and D side by side. When displaying D, make sure to rescale D's values to fill the full available dynamic range ([0,1] for float, or [0,255] for uint8). You can do this with the following operation: (D - min(D))/(max(D) - min(D)).

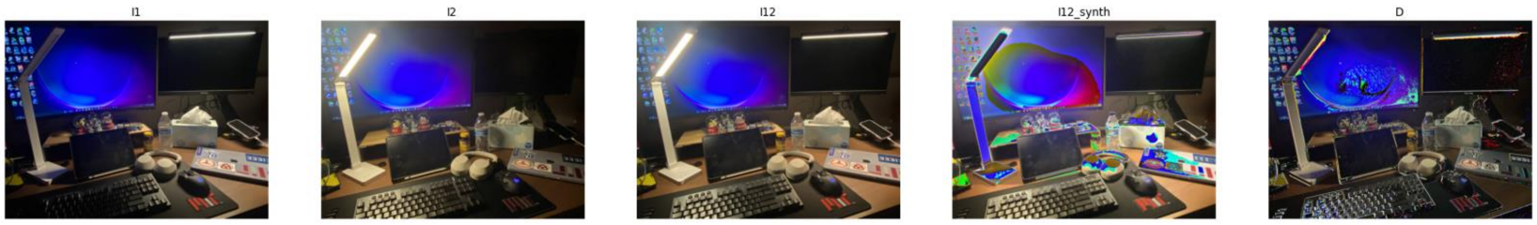


Figure 24. I1, I2, I12, I12_synth and D images

d. How good is your synthetic image compared to the real one? Where do they differ the most?

From my comparison of the synthetic image I12_synth to the real image I12, I observed that the I12_synth appears to be excessively bright. Additionally, the difference image D, which was obtained by subtracting I12_synth from I12, appears to have some noise and appears to be darker than the real image. This discrepancy is likely due to inaccuracies or noise present in the input images I1 and I2, which were used to create the synthetic image. However, if I1 and I2 accurately represent the lighting conditions for each lamp, then the synthetic image I12_synth should be a close approximation of the real image I12.

In conclusion, the synthetic image I12_synth appears to be too bright compared to the real image I12, and the difference image D highlights areas where the synthetic image deviates from the real image.