Ilan Sapoznikov

Liad Arvatz

Image Processing: Assignment #4

Problem 1- Understanding Fourier:

a) Comparing the two methods for scaling an image:

In the majority of cases, geometric operations combined with interpolation are considered the most effective and preferred method for scaling pixel coordinates. Geometric operations directly adjust the pixel coordinates, preserving the image's sharpness and intricate details. Spatial domain interpolation is easier to implement and compute, requiring less computational power, and the code is simpler to write and understand. Since these operations do not involve complex matrix transformations, they are less likely to introduce artifacts or exhibit unexpected behaviour. This method is also more user-friendly and intuitive for end users, making it easier to operate when scaling an image.

Conversely, altering and subsequently reversing the Fourier transformation can introduce artifacts and distortions, compromising the image quality, particularly if executed without precision. Although this method offers benefits for images with repetitive patterns, which can be accurately represented in the frequency domain, it is generally less suitable for images with complex details or sharp boundaries.

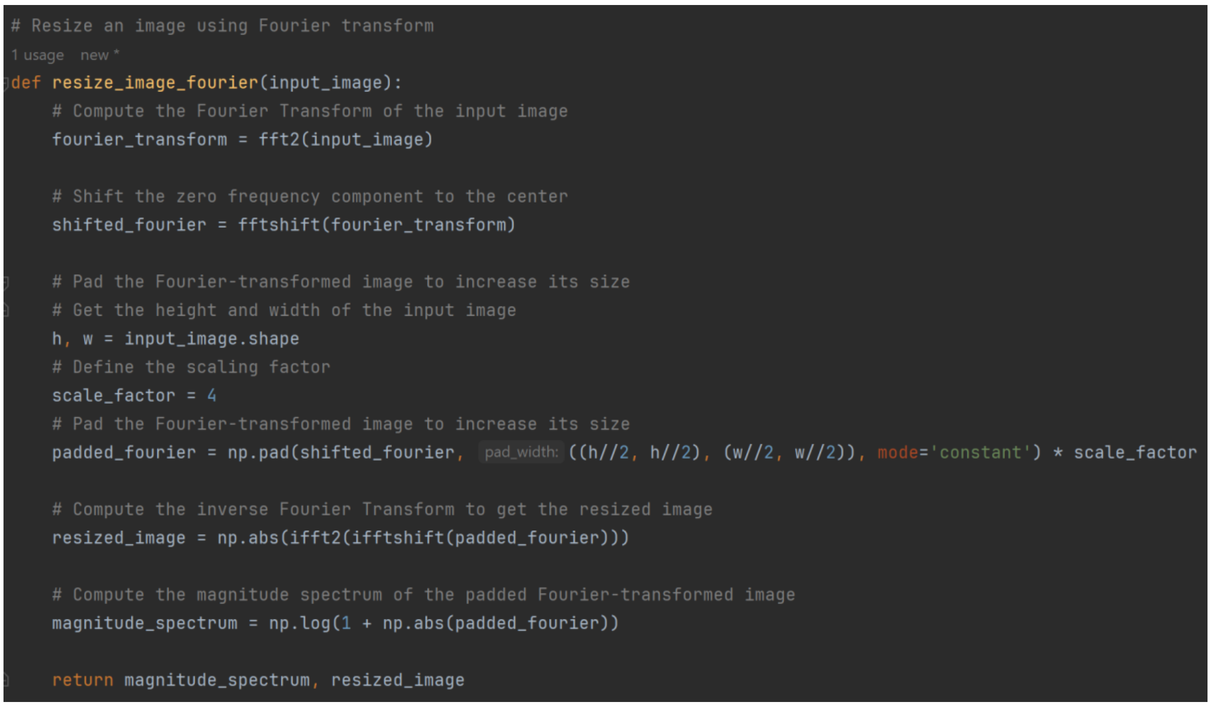
b) Uniqueness of Fourier Transform for each image:

The distinctiveness of the Fourier transformation for each image can be attributed to its encoding of the image in terms of waves characterized by directions and amplitudes. Given an image x, any rotation or translation of it will cause these waves to change direction. Similarly, any change in colour will alter the amplitude of these waves. Therefore, any combination of these changes necessitates a modification in the linear combination of sines and cosines required to encode the image. When we refer to the uniqueness of the Fourier transformation, we imply that no two distinct images share the exact same Fourier transformation. This distinctiveness stems from the transformation's ability to break down the image into its constituent frequency components. Each image encompasses a unique blend of frequencies that define its features, such as edges, textures, and colors, resulting in distinct Fourier transformations.

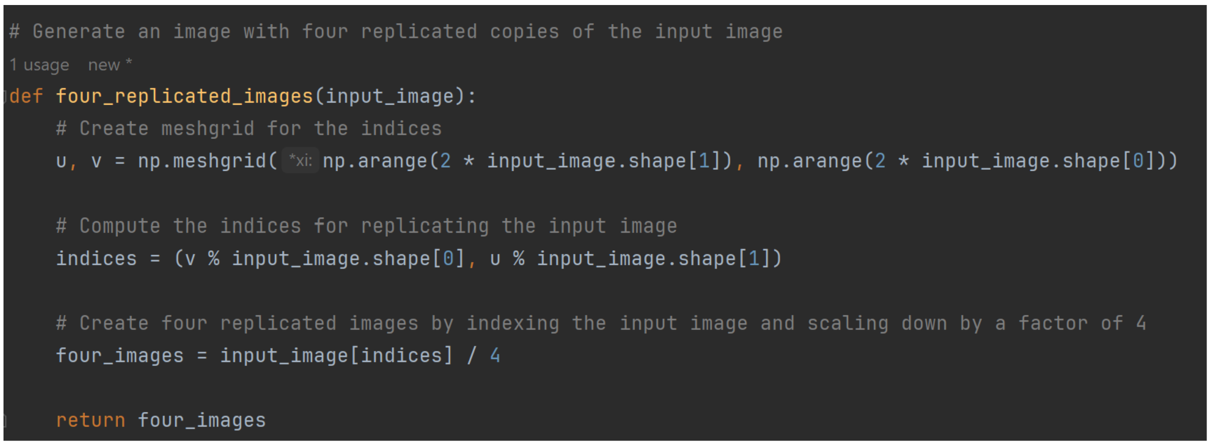
Even if two images appear identical to the naked eye, subtle differences in their frequency compositions can lead to distinct Fourier transformations. Essentially, the Fourier transformation serves as a unique identifier.

In conclusion, geometric scaling with interpolation is generally favoured for its precision, computational efficiency, and adaptability in preserving image quality during scaling. Meanwhile, the individuality of the Fourier transform for each image arises from the unique frequency components and the linear nature of the transform, ensuring that each image has a distinct representation in the frequency domain.

Problem 2- Scaling the Zebra:



```
# Resize an image using Fourier transform
1 usage  new *
def resize_image_fourier(input_image):
    # Compute the Fourier Transform of the input image
    fourier_transform = fft2(input_image)

    # Shift the zero frequency component to the center
    shifted_fourier = fftshift(fourier_transform)

    # Pad the Fourier-transformed image to increase its size
    # Get the height and width of the input image
    h, w = input_image.shape
    # Define the scaling factor
    scale_factor = 4
    # Pad the Fourier-transformed image to increase its size
    padded_fourier = np.pad(shifted_fourier,  pad_width: ((h//2, h//2), (w//2, w//2)), mode='constant') * scale_factor

    # Compute the inverse Fourier Transform to get the resized image
    resized_image = np.abs(ifft2(ifftshift(padded_fourier)))

    # Compute the magnitude spectrum of the padded Fourier-transformed image
    magnitude_spectrum = np.log(1 + np.abs(padded_fourier))

    return magnitude_spectrum, resized_image
```



```
# Generate an image with four replicated copies of the input image
1 usage  new *
def four_replicated_images(input_image):
    # Create meshgrid for the indices
    u, v = np.meshgrid( *xi: np.arange(2 * input_image.shape[1]), np.arange(2 * input_image.shape[0]))

    # Compute the indices for replicating the input image
    indices = (v % input_image.shape[0], u % input_image.shape[1])

    # Create four replicated images by indexing the input image and scaling down by a factor of 4
    four_images = input_image[indices] / 4

    return four_images
```

Code explanation:

The provided code processes a grayscale image of a zebra using Fourier transform-based techniques and visualizes the original and processed images and their Fourier spectra using matplotlib.

The “**resize_image_fourier**” function is defined to resize the input grayscale image using Fourier transform. Within this function, the Fourier transform of the input image is computed (using fft2) and its zero frequency component is shifted to the center (using fftshift), resulting in “shifted_fourier”.

After obtaining the centered Fourier transformation, the function initializes a padded image of size (2*h, 2*w) based on the dimensions of the original image. The coordinates of the image center are computed and used to position “shifted_fourier” at the center of the padded image.

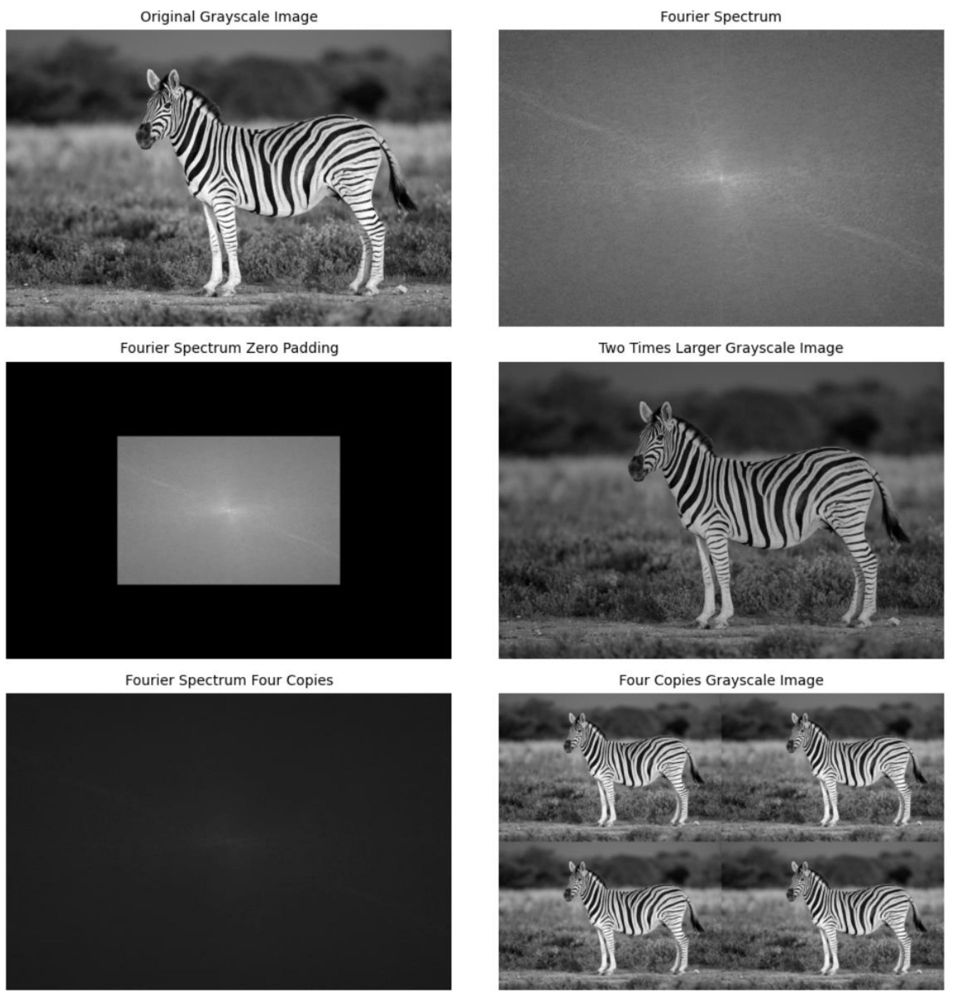
Next, the inverse FFT (using ifft2) is applied to the padded image to obtain the resized zebra image. Since the image is now padded with 4 times more pixels but has the same average (simply scaled up), the inverse operation attenuates the "energy" or intensity of each pixel. Consequently, to compensate for this loss of pixel intensity, the resized image needs to be multiplied by 4 before performing the inverse.

Lastly, the function returns the resized image (after multiplication) and the magnitude spectrum of the padded Fourier-transformed image, which is essentially the padded Fourier-transformed image before the inverse.

The “**four_replicated_images**” function is defined to generate four replicated copies of the input image. A meshgrid of indices is created to generate a 2x2 grid, and the indices are computed to replicate the input image. Four replicated images are created by indexing the input image and scaling down by a factor of 4. An array containing the four replicated images is returned.

In the main part of the code we display the images as described in the assignment instructions.

The Result:


Original Grayscale Image
Fourier Spectrum
Fourier Spectrum Zero Padding
Two Times Larger Grayscale Image
Fourier Spectrum Four Copies
Four Copies Grayscale Image

Comparison between the Scaling Methods:

1) Zero Padding:

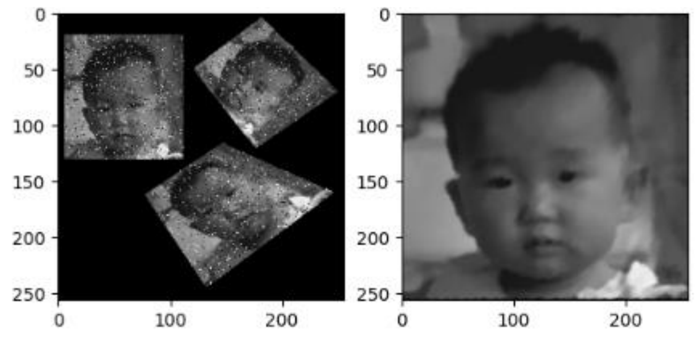
In the Fourier domain, zero padding involves adding zeros around the Fourier-transformed image to increase its size. This method does not alter the frequency content of the image but increases the resolution in the spatial domain. By padding the Fourier-transformed image with zeros, the original image remains unchanged, and we essentially obtain a larger version of the same image. However, since the image is simply scaled up, the pixel intensity or "energy" of each pixel is attenuated. To compensate for this loss of intensity, the resized image needs to be multiplied by the scaling factor (in this case, 4) before performing the inverse Fourier transform.

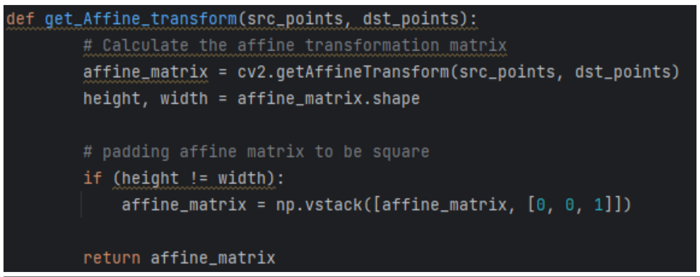
2) Interlacing (Replication):

In the replication or interlacing method, the input image is replicated multiple times to increase its size. In the frequency domain, this method introduces changes to the higher frequencies by making each frequency component twice its size. The Fourier transform views the image as an infinite wave, and by interlacing and making each wave half its size, the image is effectively shrunk relative to the window size and filled with the repeated image. When the Fourier transform is computed on the interlaced images, the resulting spectrum will have altered frequency components due to the repeated and resized images.

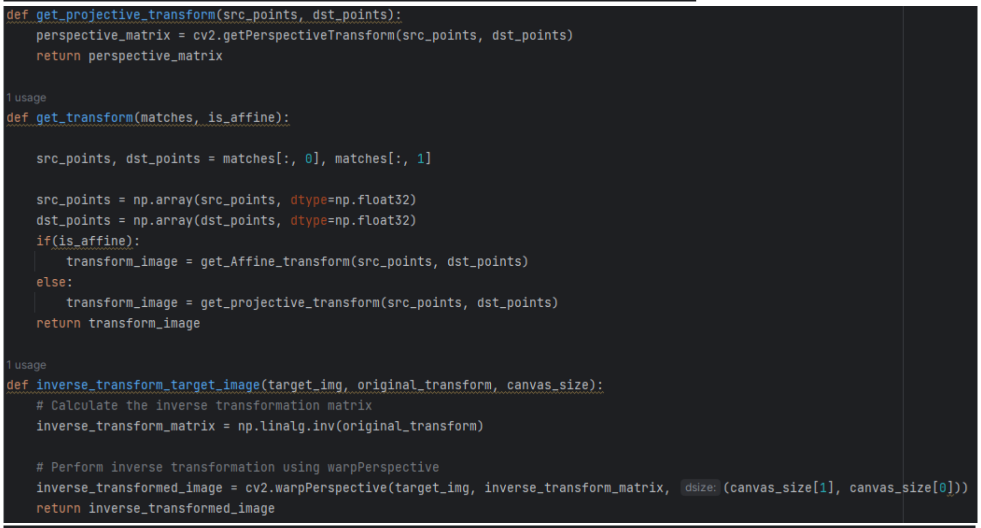
Problem 3- Fix Me!:

Image 1:

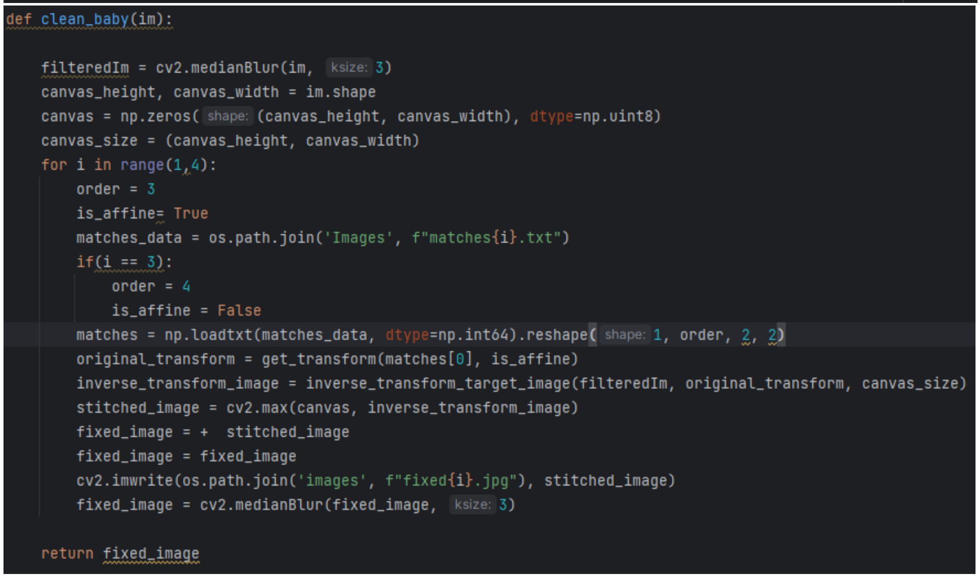




```
def get_Affine_transform(src_points, dst_points):
        # Calculate the affine transformation matrix
        affine_matrix = cv2.getAffineTransform(src_points, dst_points)
        height, width = affine_matrix.shape

        # padding affine matrix to be square
        if (height != width):
            affine_matrix = np.vstack([affine_matrix, [0, 0, 1]])

        return affine_matrix
```



```
def get_projective_transform(src_points, dst_points):
    perspective_matrix = cv2.getPerspectiveTransform(src_points, dst_points)
    return perspective_matrix

1 usage
def get_transform(matches, is_affine):

    src_points, dst_points = matches[:, 0], matches[:, 1]

    src_points = np.array(src_points, dtype=np.float32)
    dst_points = np.array(dst_points, dtype=np.float32)
    if(is_affine):
        transform_image = get_Affine_transform(src_points, dst_points)
    else:
        transform_image = get_projective_transform(src_points, dst_points)
    return transform_image

1 usage
def inverse_transform_target_image(target_img, original_transform, canvas_size):
    # Calculate the inverse transformation matrix
    inverse_transform_matrix = np.linalg.inv(original_transform)

    # Perform inverse transformation using warpPerspective
    inverse_transformed_image = cv2.warpPerspective(target_img, inverse_transform_matrix, dsize: (canvas_size[1], canvas_size[0]))
    return inverse_transformed_image
```



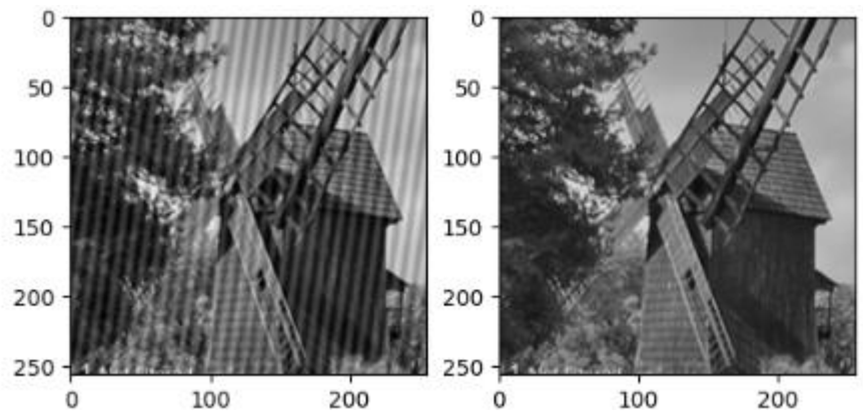
```
def clean_baby(im):

    filteredIm = cv2.medianBlur(im, ksize: 3)
    canvas_height, canvas_width = im.shape
    canvas = np.zeros( shape: (canvas_height, canvas_width), dtype=np.uint8)
    canvas_size = (canvas_height, canvas_width)
    for i in range(1,4):
        order = 3
        is_affine= True
        matches_data = os.path.join('Images', f"matches{i}.txt")
        if(i == 3):
            order = 4
            is_affine = False
        matches = np.loadtxt(matches_data, dtype=np.int64).reshape( shape: 1, order, 2, 2)
        original_transform = get_transform(matches[0], is_affine)
        inverse_transform_image = inverse_transform_target_image(filteredIm, original_transform, canvas_size)
        stitched_image = cv2.max(canvas, inverse_transform_image)
        fixed_image = +  stitched_image
        fixed_image = fixed_image
        cv2.imwrite(os.path.join('images', f"fixed{i}.jpg"), stitched_image)
        fixed_image = cv2.medianBlur(fixed_image, ksize: 3)

    return fixed_image
```

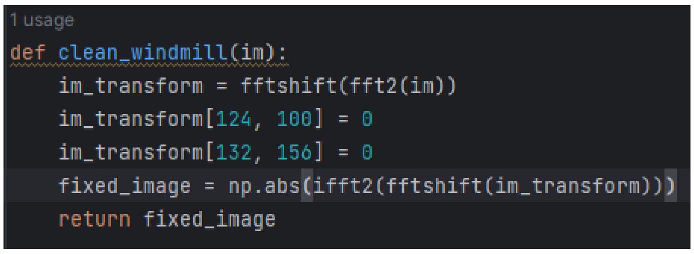
The first thing we did was to remove the salt & pepper noise using a median blur filter. As in Assignment 2, we performed geometric operations to reshape the three images and fit them onto a black canvas. The two upper images were reshaped through Affine transformation, while the bottom image underwent a Projective transformation. For all three images, we aligned the points on their edges with the edges of the entire image. Then, we stitched all the transformed images onto a black empty canvas.

Each image in the given canvas after transformation:

image 1: image 2: image 3:

Image 2:



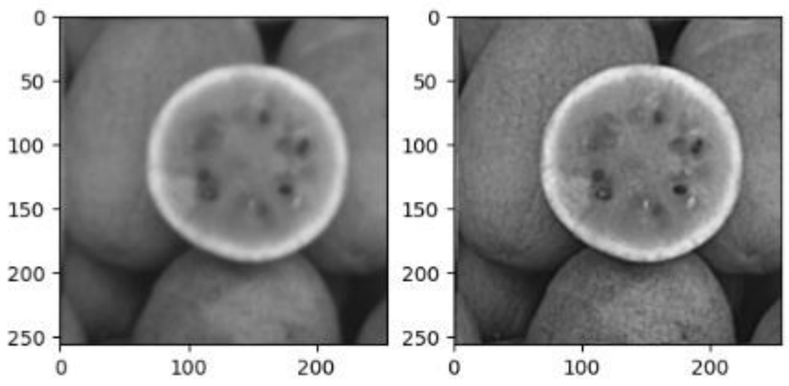


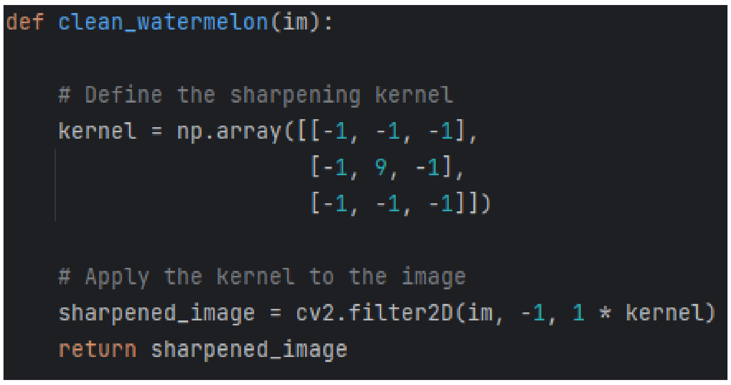
1 usage

```python
def clean_windmill(im):
    im_transform = fftshift(fft2(im))
    im_transform[124, 100] = 0
    im_transform[132, 156] = 0
    fixed_image = np.abs(ifft2(fftshift(im_transform)))
    return fixed_image
```

We identified that the noise was frequency-based, prompting us to shift our focus to the frequency domain of the image. We then located the two brightest points responsible for this noise, which we found to be at coordinates (124, 100) and (132, 156). Following what we learned in the lecture, we set the values of these points to zero and returned to the image domain.

Image 3:



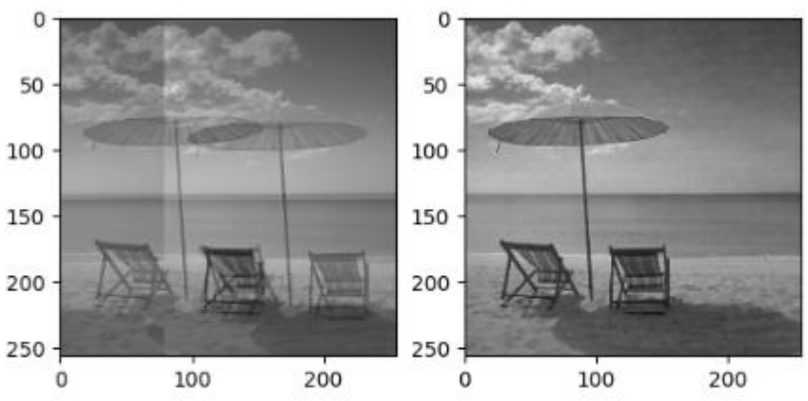


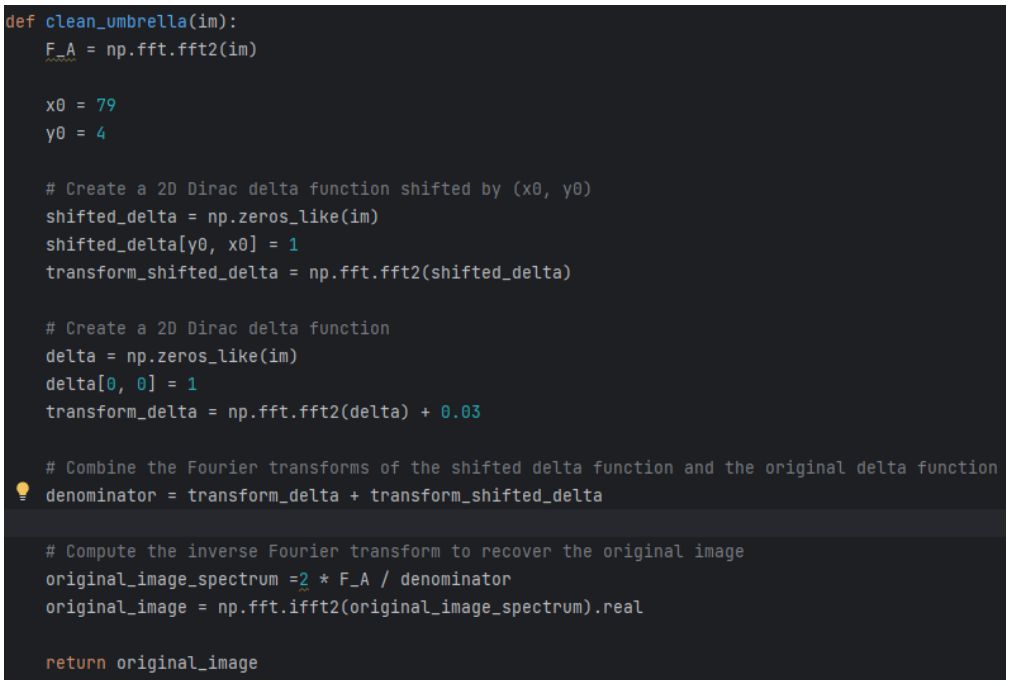
```
def clean_watermelon(im):

    # Define the sharpening kernel
    kernel = np.array([[-1, -1, -1],
                       [-1, 9, -1],
                       [-1, -1, -1]])

    # Apply the kernel to the image
    sharpened_image = cv2.filter2D(im, -1, 1 * kernel)
    return sharpened_image
```

We noticed that the image appeared more blurred, but the edges were preserved.

So we applied a sharpening kernel, which enhances edges by emphasizing the differences in intensity between neighboring pixels. The central pixel is multiplied by a larger weight (9), making it the dominant factor, while the surrounding pixels are multiplied by smaller weights (-1), reducing their influence. This difference in weighting helps accentuate the edges in the image.

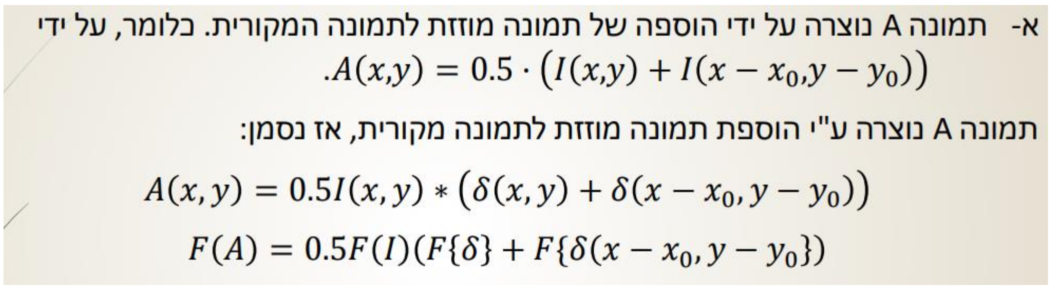
Image 4:





```python
def clean_umbrella(im):
    F_A = np.fft.fft2(im)

    x0 = 79
    y0 = 4

    # Create a 2D Dirac delta function shifted by (x0, y0)
    shifted_delta = np.zeros_like(im)
    shifted_delta[y0, x0] = 1
    transform_shifted_delta = np.fft.fft2(shifted_delta)

    # Create a 2D Dirac delta function
    delta = np.zeros_like(im)
    delta[0, 0] = 1
    transform_delta = np.fft.fft2(delta) + 0.03

    # Combine the Fourier transforms of the shifted delta function and the original delta function
    denominator = transform_delta + transform_shifted_delta

    # Compute the inverse Fourier transform to recover the original image
    original_image_spectrum =2 * F_A / denominator
    original_image = np.fft.ifft2(original_image_spectrum).real

    return original_image
```

We noticed that the image is the sum of the real image and a shifted version of the real image. We measured the distance between two identical points (the upper edge of the umbrella) and assigned the vertical axis distance to x0 and the horizontal axis distance to y0. We then transitioned to the frequency domain and applied the calculation method we learned in Tutorial 7:



א- תמונה A נוצרה על ידי הוספה של תמונה מוזזת לתמונה המקורית. כלומר, על ידי $.A(x,y) = 0.5 \cdot \big(I(x,y) + I(x - x_0,y - y_0)\big)$

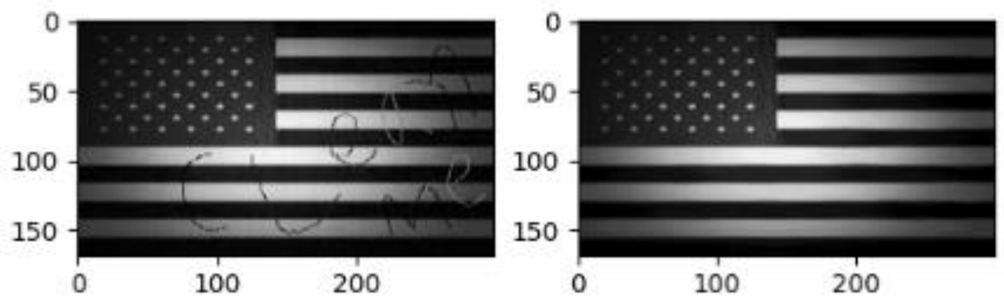
תמונה A נוצרה ע"י הוספת תמונה מוזזת לתמונה מקורית, אז נסמן:

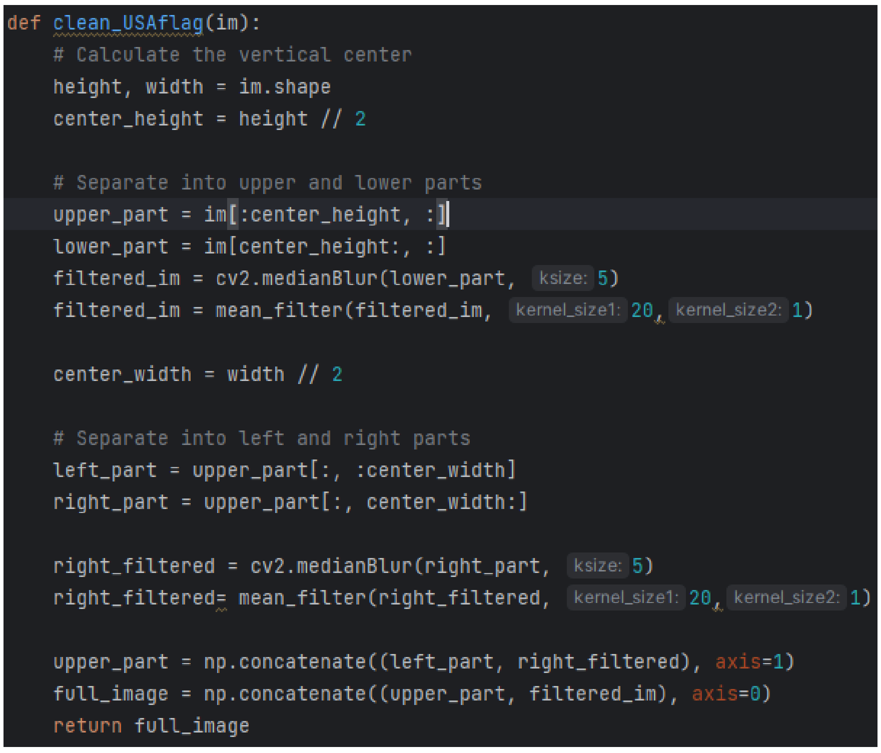
$$A(x,y) = 0.5I(x,y) * \big(\delta(x,y) + \delta(x - x_0, y - y_0)\big)$$

$$F(A) = 0.5F(I)(F\{\delta\} + F\{\delta(x - x_0, y - y_0\})$$

We added 0.03 to denominator to avoid division by 0.

Image 5:



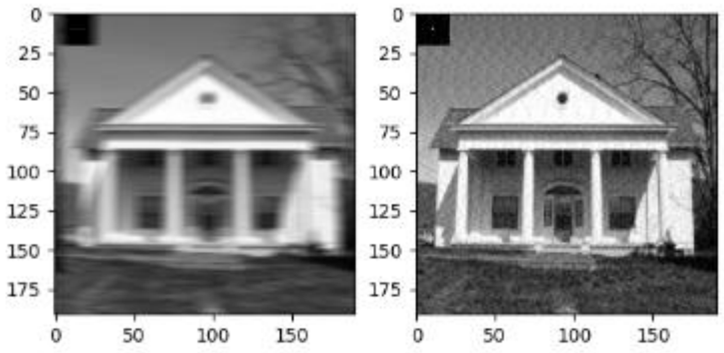


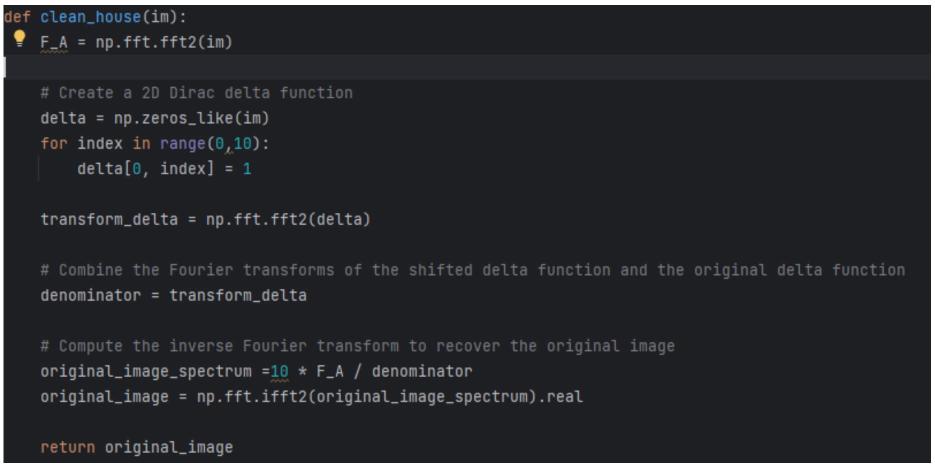
```python
def clean_USAflag(im):
    # Calculate the vertical center
    height, width = im.shape
    center_height = height // 2

    # Separate into upper and lower parts
    upper_part = im[:center_height, :]
    lower_part = im[center_height:, :]
    filtered_im = cv2.medianBlur(lower_part, ksize: 5)
    filtered_im = mean_filter(filtered_im, kernel_size1: 20, kernel_size2: 1)

    center_width = width // 2

    # Separate into left and right parts
    left_part = upper_part[:, :center_width]
    right_part = upper_part[:, center_width:]

    right_filtered = cv2.medianBlur(right_part, ksize: 5)
    right_filtered= mean_filter(right_filtered, kernel_size1: 20, kernel_size2: 1)

    upper_part = np.concatenate((left_part, right_filtered), axis=1)
    full_image = np.concatenate((upper_part, filtered_im), axis=0)
    return full_image
```

We noticed that the left upper side (the part with the stars) was delicate, so we decided to divide the image into two equal parts: an upper part and a bottom part. The lower part of the picture contained the stripes of the flag, and it was relatively easy to remove the noise using a median blur with a 5x5 kernel, followed by a mean filter with a 20x1 kernel.

Afterward, we further divided the upper part into two sections: left and right. We applied the same filtering process to the right part, which also included the flag's stripes. Finally, we concatenated the left part with the right part and then merged the lower part with the upper part before returning the complete image.

Image 6:

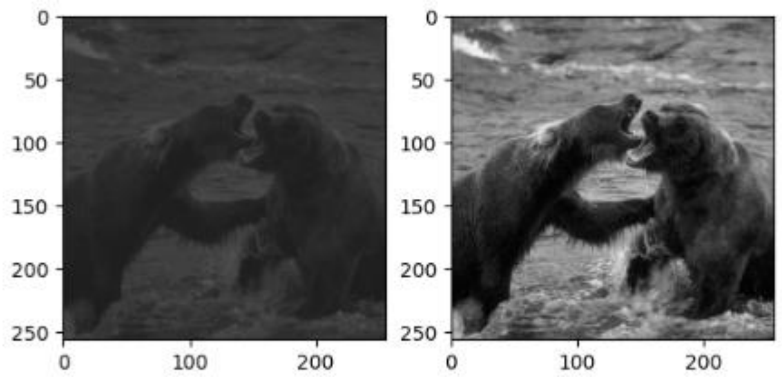


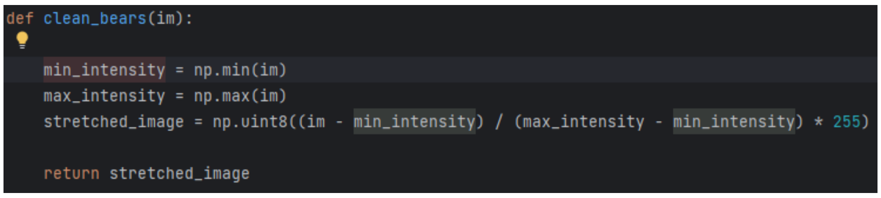


```python
def clean_house(im):
    F_A = np.fft.fft2(im)

    # Create a 2D Dirac delta function
    delta = np.zeros_like(im)
    for index in range(0,10):
        delta[0, index] = 1

    transform_delta = np.fft.fft2(delta)

    # Combine the Fourier transforms of the shifted delta function and the original delta function
    denominator = transform_delta

    # Compute the inverse Fourier transform to recover the original image
    original_image_spectrum =10 * F_A / denominator
    original_image = np.fft.ifft2(original_image_spectrum).real

    return original_image
```

Using the same method as we used for the umbrella image, we determined that this image is the sum of 10 shifted images. Consequently, we initialized the first 10 cells in delta to be 1 and the rest to be 0. This is equivalent to adding 10 shifted matrices together.

Image 7:





```
def clean_bears(im):

    min_intensity = np.min(im)
    max_intensity = np.max(im)
    stretched_image = np.uint8((im - min_intensity) / (max_intensity - min_intensity) * 255)

    return stretched_image
```

We performed contrast stretching or normalization. By identifying the minimum and maximum intensity values, we stretched the intensity range to cover the full [0, 255] range and returned the resulting stretched image. This process enhances the contrast of the image.