Problem 1 – Understanding Fourier (12 points):

Section a:

The choice between **geometric operations** with interpolation and **Fourier transform**-based scaling depends on the specific use case and the requirements of the task. Geometric operations with interpolation are widely preferred for most applications due to their straightforward implementation and computational efficiency. They allow scaling images with commonly used interpolation methods such as bilinear or bicubic, producing good results for small to moderate scaling factors. However, this method can struggle when scaling images with significant size changes or high-frequency details, as interpolation may introduce artifacts and fail to preserve intricate patterns.

On the other hand, Fourier transform-based scaling excels in preserving frequency details and handling non-integer scaling factors. By working in the frequency domain, it offers precise control over high-frequency components, which can be advantageous in tasks requiring the preservation of fine image details. However, this method is computationally expensive and more complex to implement, making it less practical for large images or real-time applications.

In most practical scenarios, especially in real-time systems where computational efficiency and simplicity are key, geometric operations with interpolation are the better choice. They provide satisfactory results for a wide range of tasks while being faster and easier to implement. Fourier transform-based scaling, while useful in niche scenarios where frequency preservation is critical, is generally not the preferred approach for routine scaling tasks due to its complexity and cost.

Section b:

From the definition, we know that the 2D Fourier transform of a function $f(x, y)$ is given as:

$$\boldsymbol{F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\cdot e^{-i2\pi(ux+vy)}\,dx\,dy}$$

Similarly, for $g(x, y)$, its Fourier transform is:

$$\boldsymbol{G(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(x,y)\cdot e^{-i2\pi(ux+vy)}\,dx\,dy}$$

We are given that $F(u, v) = G(u, v)$, This implies that the integrals for $f(x, y)\ and\ g(x, y)$ are equal for all values of u and v.

Using the inverse Fourier transform, we can reconstruct $f(x, y)$ from $F(u, v)$:

$$\boldsymbol{f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)\cdot e^{i2\pi(ux+vy)}\,du\,dv}$$

Similarly, for $g(x, y)$:

$$\boldsymbol{g(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} G(u,v)\cdot e^{i2\pi(ux+vy)}\,du\,dv}$$

Since $F(u, v)$ = $G(u, v)$, substituting this into the inverse Fourier transform gives:

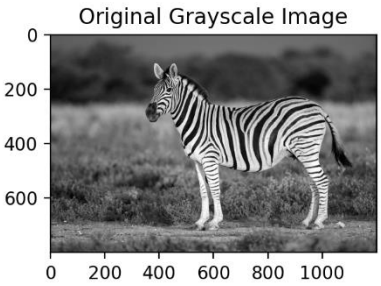
$$f(x, y) = g(x, y)$$

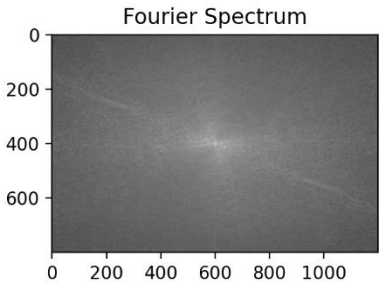
Thus, we have proven mathematically that if $F(u, v)$ = $G(u, v)$, then $f(x, y) = g(x, y)$.

Problem 2 – Scaling the Zebra (18 points)

Section a:

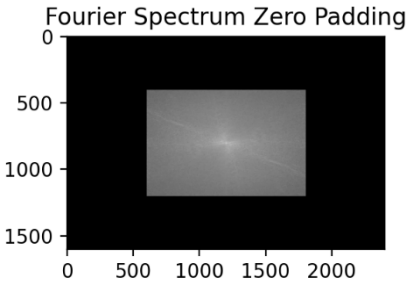
we compute the Fourier Transform of the zebra image using a 2D Fourier Transform (fft2) to represent the image in the frequency domain. The zero-frequency component is shifted to the center using fftshift for better visualization, and the magnitude spectrum is calculated using a logarithmic scale to enhance visibility

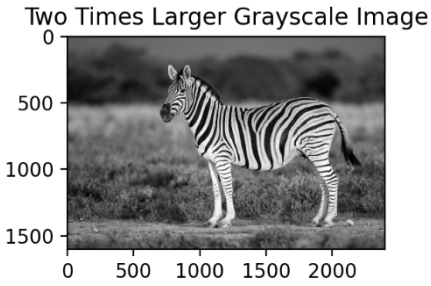




section b:

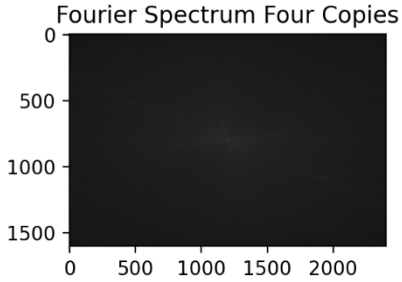
Here, we scaled the image using **zero padding** in the Fourier domain. Zero padding was applied to double the dimensions of the original Fourier transform. The process involved embedding the Fourier transform into the center of a larger zero-padded array. This increases the spatial resolution of the reconstructed image, effectively scaling the image. We then performed an inverse Fourier transform to retrieve the scaled grayscale image. The brightness of the scaled image was corrected by multiplying it by a factor of 4, ensuring consistent intensity.

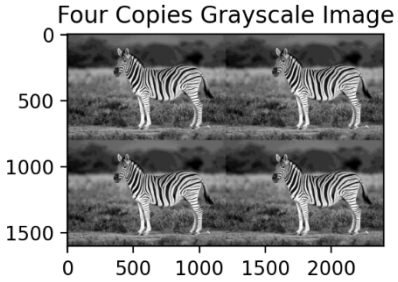




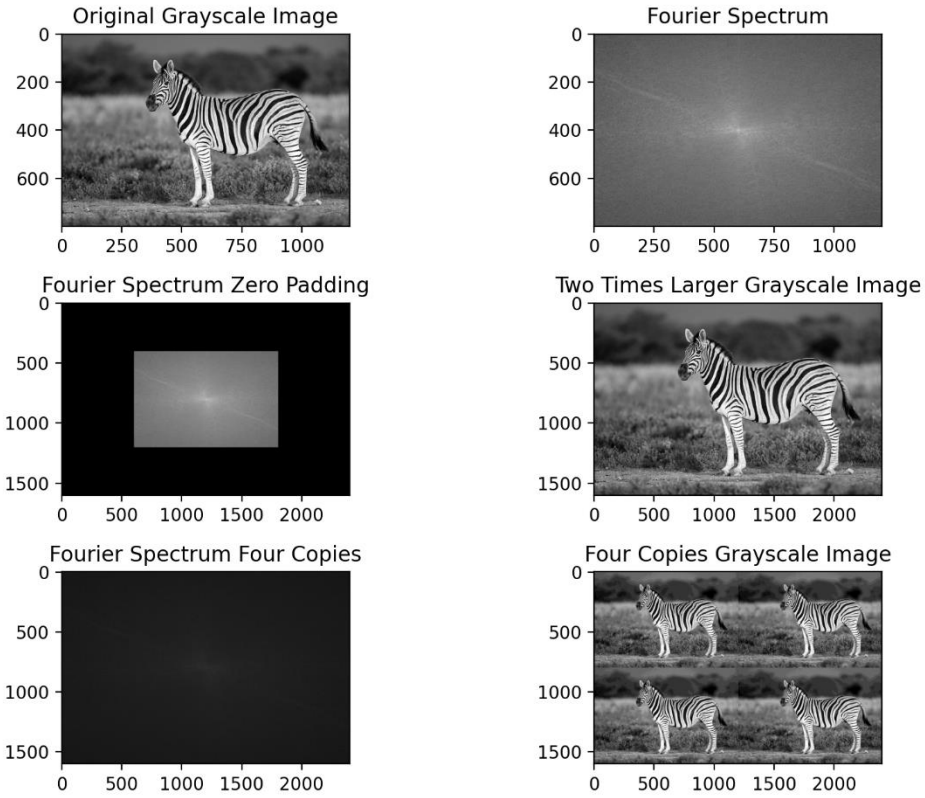
Section c:

Here we implemented the Fourier Transform Scaling Formula to generate an image containing four copies of the original zebra image. The formula ensured that the Fourier coefficients were redistributed correctly to create a larger Fourier transform. We iterated through even indices in the new transform and mapped them to corresponding points in the original Fourier transform. The final scaled image was reconstructed by applying the inverse Fourier transform and normalized for proper visualization. This method produced four replicas of the original image, evenly distributed within the larger output.





Final Scaled Output:



The two methods approach the concept of scaling an image differently by manipulating the Fourier transform. While both techniques involve frequency domain operations, they result in distinct outcomes in the spatial domain, as detailed below:

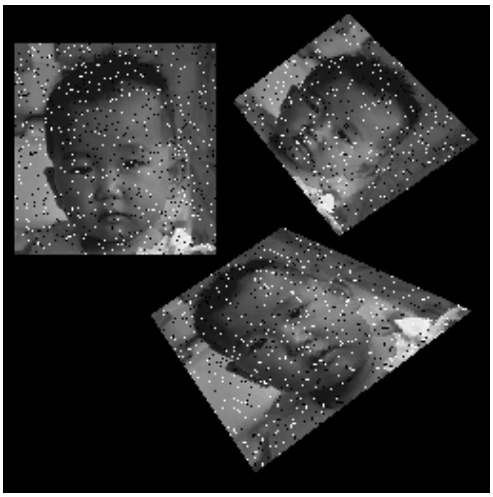
1. **Zero Padding Scaling:**
   This approach enlarges the frequency spectrum by a factor of 2 along both dimensions. The Fourier transform of the original image is embedded into the center of a larger zero-padded array, effectively stretching the frequency components. When applying the inverse Fourier transform, the resulting image appears twice the size of the original, maintaining higher spatial frequency details, which makes it sharper and larger.

2. **Four Copies Using Fourier Scaling Formula:**
   This method replicates the frequency information of the original image four times, creating duplicates in the spatial domain. By distributing each pixel of the Fourier transform into four corresponding locations within a larger Fourier domain, it ensures the image repeats in a grid pattern. After the inverse Fourier transform, we observe four smaller, identical copies of the original image arranged within the new image, resulting in a visually distinctive pattern.

Problem 3 – Fix me! (70 points):

Baby:

original image:



Cleaned Image:

How we fixed the image:

We extracted the three noisy images using perspective transformations and then combined them using a median filter. This approach removed the noise while preserving the original details in the reconstructed image.

Code Snippet:

```python
def clean_baby(im):
    denoised_im = cv2.medianBlur(im, ksize=3)
    # Perform perspective transformations
    transformed_images = []
    points = get_transformation_points()
    for source, destination in points:
        transformation_matrix = cv2.getPerspectiveTransform(source, destination)
        transformed_image = cv2.warpPerspective(
            denoised_im, transformation_matrix, (256, 256), flags=cv2.INTER_CUBIC
        )
        transformed_images.append(transformed_image)

    # Combine the transformed images using a median filter
    combined_image = np.median(np.array(transformed_images), axis=0)
    return combined_image.astype(np.uint8)
```

Code Explanation:

This function removes noise from the image by:

1. **Denoising**:
    - Applies a median blur to reduce random noise while preserving edges.
2. **Perspective Transformations**:
    - Extracts three images from the original noisy image using predefined source and destination points.
    - This ensures the proper alignment of each individual image.
3. **Combining Images**:
    - Uses a median filter to combine the extracted images, effectively removing the noise while retaining details.

This process effectively reconstructs a clear image by leveraging the redundancy and alignment of the three noisy images.

bears:

original image:

Cleaned Image:

How we fixed the image:

This image was enhanced by normalizing pixel intensities to the full range (0–255) for uniform brightness, followed by CLAHE for adaptive contrast enhancement, which improved visibility while preserving local details and preventing overexposure or underexposure.

Code Snippet:

```python
def clean_bears(im):
    normalized_im = cv2.normalize(im, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8U)
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    enhanced_im = clahe.apply(normalized_im)

    return enhanced_im
```

Code Explanation:

This function enhances low-light images by combining two techniques:

1. **Normalization**: Ensures the image's pixel intensity values span the full range (0–255), brightening the overall image uniformly.
2. **CLAHE**: Adjusts contrast locally, preserving details in darker regions and preventing overexposure. The clipLimit=2.0 parameter controls the degree of contrast enhancement, while the grid size ensures adaptive enhancement for different parts of the image.

house:

original image:

Cleaned Image:

How we fixed the image:

This image had motion blur caused by averaging shifted images. We used the Fourier Transform to analyze its frequency components, applied a custom mask to suppress noise frequencies, and ensured stability with a threshold. The corrected data was transformed back using the inverse Fourier Transform and normalized for better visualization.

Code Snippet:

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

    def create_mask(rows, cols, threshold=0.02):
        base_mask = np.zeros((rows, cols))
        base_mask[0, :10] = 0.1
        mask_fft = np.fft.fft2(base_mask)
        return np.where(abs(mask_fft) < threshold, 1, mask_fft)

    mask = create_mask(*im.shape)

    corrected_fft_image = fft_image / mask

    cleaned_image = np.fft.ifft2(corrected_fft_image)
    cleaned_image = np.abs(cleaned_image)

    cleaned_image = (255 * (cleaned_image - cleaned_image.min()) /
                     (cleaned_image.max() - cleaned_image.min()))

    return cleaned_image.astype(np.uint8)
```

Code Explanation:

This function removes motion blur using the following steps:

1. **Frequency Analysis:**
    - The input image is transformed into the frequency domain to analyze its components.
2. **Mask Application:**
    - A frequency mask is created to suppress unwanted frequencies responsible for the motion blur.
3. **Inverse Transform:**
    - The cleaned frequency data is transformed back into the spatial domain, reconstructing the deblurred image.
4. **Normalization:**
    - The cleaned image is normalized to enhance its contrast and ensure it is in the correct intensity range (0–255).

umbrella:

original image:

Cleaned Image:

How we fixed the image:

This image had noise caused by averaging the original image and a shifted version. To fix it, we used the Fourier Transform to analyze the frequency components, created a custom frequency mask to suppress unwanted noise frequencies, and applied it while ensuring numerical stability. The corrected frequency data was transformed back using the inverse Fourier Transform, and the result was normalized for better visualization.

Code Snippet:

```python
def clean_umbrella(im):
    freq_domain = np.fft.fft2(im)
    # Build the mask
    def generate_mask(shape):
        mask = np.zeros(shape)
        mask[4, 79] = 0.5
        mask[0, 0] = 0.5
        return mask

    mask = generate_mask(im.shape)
    mask_freq = np.fft.fft2(mask)
    # avoid division errors
    safe_mask = np.where(abs(mask_freq) < 0.01, 1, mask_freq)
    # Filter the image in the frequency domain
    filtered_freq = freq_domain / safe_mask
    cleaned_image = np.abs(np.fft.ifft2(filtered_freq))
    cleaned_image = (255 * (cleaned_image - cleaned_image.min()) /
                     (cleaned_image.max() - cleaned_image.min()))

    return cleaned_image.astype(np.uint8)
```

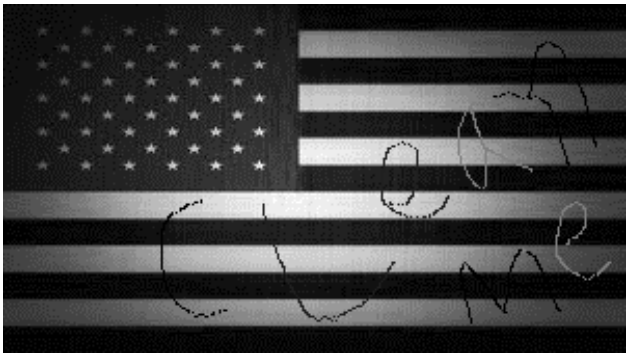
Code Explanation:

This function fixes the image by:

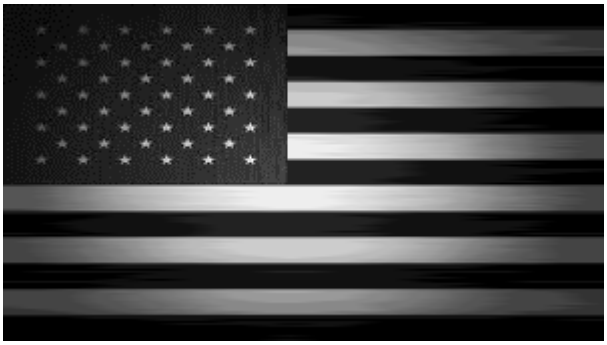
1. **Frequency Analysis**:
    - The input image is transformed into the frequency domain using the Fourier Transform to analyze its components.
2. **Mask Creation**:
    - A mask was created to suppress unwanted frequency components responsible for the noise.
    - The mask ensures stability by setting small values to a safe threshold.
3. **Noise Suppression**:
    - The noisy frequencies were filtered out in the frequency domain by dividing the Fourier-transformed image by the mask.
4. **Reconstruction**:
    - The cleaned frequency data was transformed back into the spatial domain using the inverse Fourier Transform.
5. **Normalization**:
    - The final result was normalized to enhance its contrast and ensure proper intensity scaling (0–255).

USAflag:

original image:



Cleaned Image:



How we fixed the image:

This image was cleaned using a combination of median filtering and blurring. For the lower portion (rows 90:), a median filter (1, 50) smoothed scribbles while preserving edges, followed by a blur filter (15, 1) to reduce horizontal noise. In the upper-right corner (columns 142:), the same filters were applied to remove residual scribbles while preserving the details of the stars' section.

Code Snippet:

```python
def clean_USAflag(im):
    clean_im = im.copy()
    # Apply a median filter and blur to the lower portion
    clean_im[90:, :] = median_filter(im[90:, :], size=(1, 50))
    clean_im[90:, :] = cv2.blur(clean_im[90:, :], ksize=(15, 1))
    # Apply a median filter and blur to the upper-right portion
    clean_im[:90, 142:] = median_filter(im[:90, 142:], size=(1, 50))
    clean_im[:90, 142:] = cv2.blur(clean_im[:90, 142:], ksize=(15, 1))

    return clean_im
```

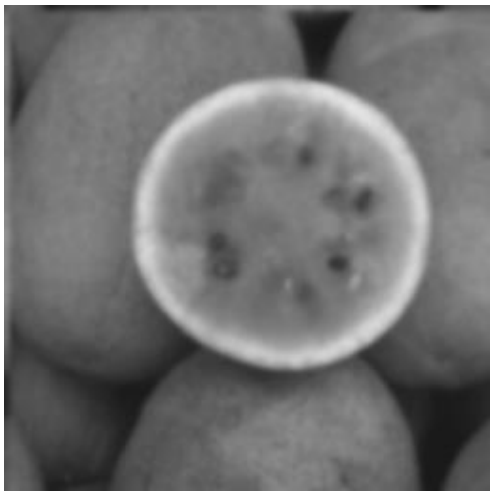
Code Explanation:

This function removes scribbles while preserving the key features of the flag:

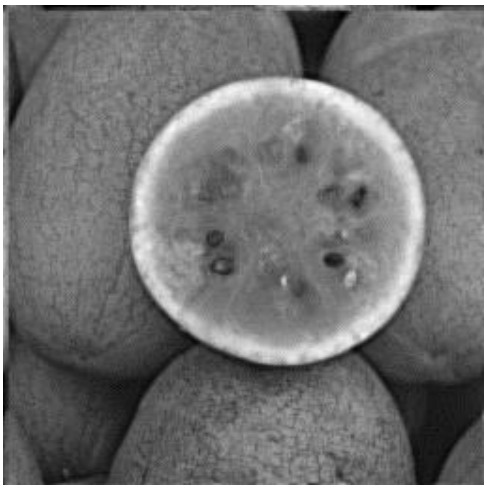
1. Lower Portion: The combination of a median filter and blur removes horizontal scribbles while maintaining the clean stripes and stars.
2. Upper Right Corner: The stars section remains intact except for the noisy right side, where the filters ensure smoothness and consistency.
3. Region-Specific Filtering: Only the affected regions are processed, leaving the rest of the flag unchanged for better detail retention.

watermelon:

original image:



Cleaned Image:



How we fixed the image:

We applied a sharpening filter using a kernel with surrounding values of -4 and a center value of 17, which reduced the influence of neighboring pixels while amplifying the central pixel's intensity. This enhanced the edges and details of the watermelon without adding artifacts.

Code Snippet:

```
def clean_watermelon(im):
    sharpening_kernel = np.array([[0, -4,  0],[-4,  17, -4],[0, -4,  0]])
    sharpened_image = cv2.filter2D(im, -1, sharpening_kernel)

    return sharpened_image
```

Code Explanation:

This function sharpens the image by applying a 3x3 kernel with a center value of 17 and surrounding values of -4. The kernel enhances edges by subtracting neighboring pixel intensities while amplifying the central pixel. The result is a sharper image with more defined edges, highlighting the details of the watermelon.

windmill:

original image:

Cleaned Image:

How we fixed the image:

We used the Fourier Transform to move the image to the frequency domain. The noise was identified as specific frequencies. These frequencies were suppressed by setting their values

to zero. The cleaned frequency domain was then transformed back into the spatial domain using the Inverse Fourier Transform, resulting in a noise-free image.

Code Snippet:

```python
def clean_windmill(im):
    freq_domain = np.fft.fft2(im)
    shifted_freq = np.fft.fftshift(freq_domain)
    noise_offsets = [(4, 28), (-4, -28)]
    center_row, center_col = im.shape[0] // 2, im.shape[1] // 2

    for offset in noise_offsets:
        row_offset, col_offset = offset
        shifted_freq[center_row + row_offset, center_col + col_offset] = 0
        shifted_freq[center_row - row_offset, center_col - col_offset] = 0

    unshifted_freq = np.fft.ifftshift(shifted_freq)
    cleaned_image = np.fft.ifft2(unshifted_freq)

    return np.abs(cleaned_image)
```

Code Explanation:

This function removes noise by transforming the image into the frequency domain using a 2D Fourier Transform. Specific noise frequencies, identified by offsets (4, 28) and (-4, -28), are suppressed by setting their values to zero. The modified frequency spectrum is then transformed back to the spatial domain using the Inverse Fourier Transform, resulting in a cleaned image.