Image Processing Assignment 2

Transformations in Spatial and Frequency Domain

NAME: SRI PRIYAN S ROLL NO: 2019103065

Intensity Transformation (Gamma Transformation)

Gamma Transformation can be used to map a narrow range of dark input values to a wider range of output values. This can make the components in the dark regions of the image visible that are otherwise hidden. The transformation function is $T(r) = cr^{\gamma}$.

```
def gamma_transform(grayscale_image, c, gamma):
    return rescale_image(c * (grayscale_image ** gamma))

def rescale_image(grayscale_image):
    max_val = np.max(grayscale_image)
    res = (grayscale_image / max_val) * 255
    return res
```

After performing the gamma transformation, the values will not be in the same range of [0, 255]. It will be shrinked or expanded depending on the value of γ. So, we have to rescale the intensity values so that they lie in the range [0, 255].

```
def gamma_mapping(n, c, gamma):
    return c * (n ** gamma)

def rescale_gamma_mapping(cur_val, max_mapped_val, start, end):
    return start + (cur_val / max_mapped_val) * (end - start)

def range_gamma_transform(grayscale_image, start, end, c, gamma):
    mask = (grayscale_image >= start) & (grayscale_image <= end)
    max_val = np.max(grayscale_image[mask])
    max_mapped_val = gamma_mapping(max_val, c, gamma)

    return np.where(
        mask,
```

```
        rescale_gamma_mapping(
            gamma_mapping(grayscale_image, c, gamma),
            max_mapped_val, start, end
        ),
        grayscale_image
    )
```

The ***gamma_transform*** function performs the Gamma Transformation for the entire image. But that doesn't give much control to us on the intensity values we want to work on. In order to solve that I have created the ***range_gamma_transform*** function that allows us to work on specific ranges of Intensities.

Orginal Image

Gamma Transformation with y = 0.3

Intensity Range [0, 100] Transformed with y = 0.2

[0, 100] with y = 0.2 and [120, 255] with y = 5

The second image is the gamma transformation of the entire original image. We can see that the details in the darker regions have become visible. But the brighter regions have gotten more bright making it feel unrealistic.

The third image shows the gamma transformation of only those pixels with intensity values in the range [0, 100] with γ = 0.2. This looks slightly better than the second image because the brighter region is left untouched.

But we can do better. The fourth image applies two gamma transformations one after the other on the input image, one with γ = 0.2 on the range [0, 100] and another one with γ = 5 on the range [120, 255]. The second one makes the bright regions less bright so that the image looks more realistic.

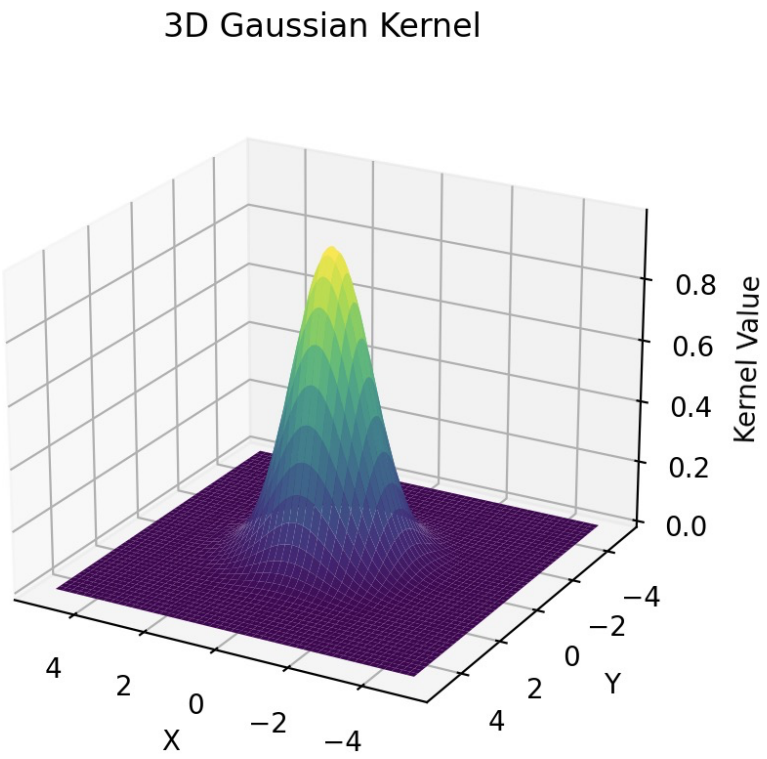
Spatial Domain Filtering

Smoothing Filter

A simple smoothing filter is the averaging filter. By taking the average of the pixels covered by the kernel, the high frequency components are being reduced. They can be used to remove the noise from images.

```
def average_filter(grayscale_image, kernel_size):
    kernel = np.ones((kernel_size, kernel_size)) / (kernel_size ** 2)
    return cv.filter2D(grayscale_image, -1, kernel)
```

The most common smoothing filter is the Gaussian Filter. The kernel is created with the values obtained from the Gaussian curve. The below plot shows how the kernel values are high towards the center and decreases as it moves away.



```
avg_image_7x7 = average_filter(image, 7)
gaussian_blur_11x11 = cv.GaussianBlur(image, (11, 11), 0)
gaussian_blur_17x17 = cv.GaussianBlur(image, (17, 17), 0)
```

Orginal Image

Averaging with 7x7 kernel

Smoothen with 11x11 Gaussian kernel

Smoothen with 17x17 Gaussian kernel

The original image has a lot of noise and we can see the noise being reduced more and more with the second, third and fourth images.

Sharpening Filter

The Laplacian filter is used to identify the edges(High Frequency Components) where the pixel intensities vary rapidly with its neighbors. After applying the Laplacian Filter, the image may contain both positive and negative values. So, we have to rescale it to be positive and occupy the range [0, 255]

```
def laplacian_filter(grayscale_image):
    kernel = np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]])
    return cv.filter2D(grayscale_image, cv.CV_64F, kernel)
```

```
def rescale_laplacian(grayscale_image):
   grayscale_image = grayscale_image - np.min(grayscale_image)
   return ((grayscale_image / np.max(grayscale_image)) *
255).astype(np.uint8)
```

Then we do a weighted sum of the actual Image and the scaled Laplacian Image to get the sharpened Image. The output can be seen below.

```
laplacian = laplacian_filter(image)
scaled_laplacian = rescale_laplacian(laplacian)
sharpened_image = cv.addWeighted(image, 1.5, scaled_laplacian, -0.5,
0)
```

Original Image

Laplacian Image

Scaled Laplacian Image

Sharpened Image

We can clearly see the fourth image is a sharpened version of the first image. The Laplacian and the Scaled Laplacian Images are also shown.

Frequency Domain Filtering

Smoothing Filter (Low Pass Filter)

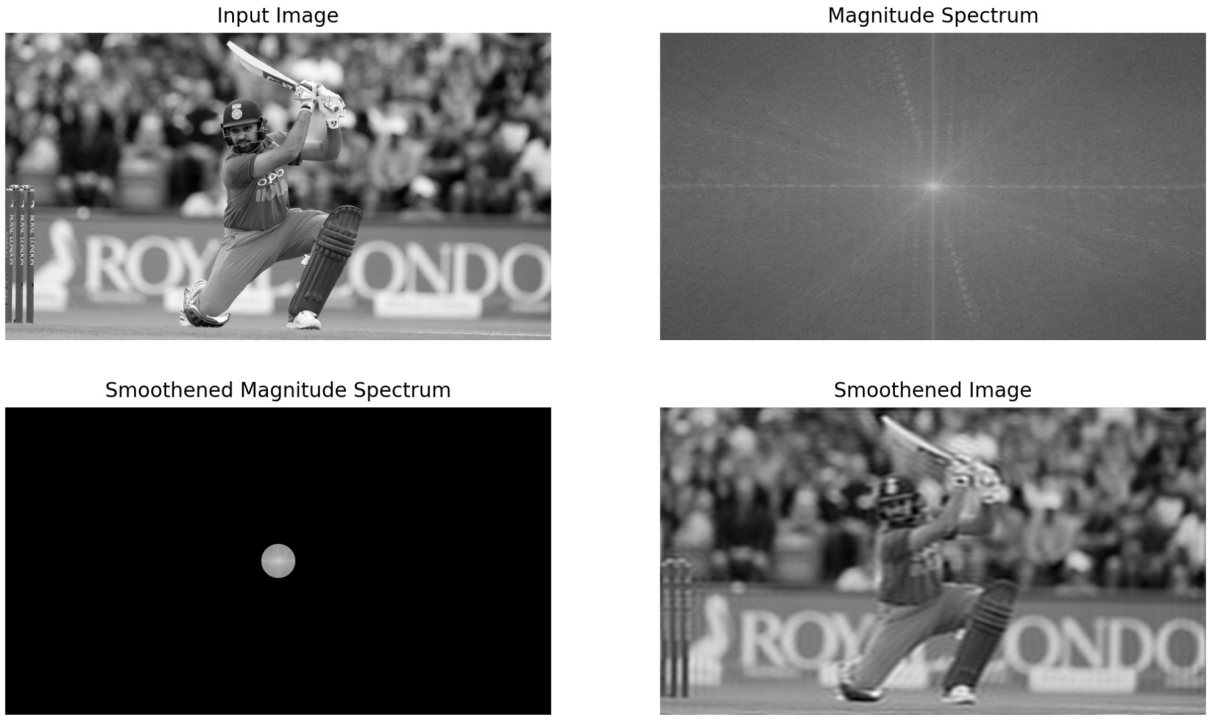
First we have to convert the image from the spatial domain to the frequency domain. I have used the Fast Fourier Transform which is an implementation of the Discrete Fourier Transform. Then we can shift the low frequency components to the center to make filtering easier. The transformed image can be visualized as the Magnitude Spectrum. The filters can be applied to the transformed image and then converted back to the spatial domain using the Inverse Fast Fourier Transform.

```
# Apply Fourier Transform to the Image
ft_image = np.fft.fft2(image)
shifted_ft_image = np.fft.fftshift(ft_image)
magnitude_spectrum = 20 * np.log(np.abs(shifted_ft_image))

# Smoothen the Transformed Image
smoothened_shifted_ft_image = low_pass_filter(shifted_ft_image,
filter_size)
smoothened_magnitude_spectrum = 20 * np.log(
   np.abs(smoothened_shifted_ft_image) + 1
)

# Inverse Fourier Transform to get the smoothened image
smoothened_ft_image = np.fft.ifftshift(smoothened_shifted_ft_image)
smoothened_image = np.real(np.fft.ifft2(smoothened_ft_image))
```

We can apply different types of smoothing filters. One way is to mask out all the high frequency components and keep only the low frequency components in the center. This is called the ideal low pass filter.

```
def ideal_low_pass_filter(shifted_ft_image, size):
   rows, cols = shifted_ft_image.shape
   c_row, c_col = rows // 2, cols // 2

   mask = np.zeros(shifted_ft_image.shape)
```

```
    u, v = np.meshgrid(np.arange(rows), np.arange(cols),
indexing='ij')
    mask[D(u, v, c_row, c_col) <= size] = 1

    return shifted_ft_image * mask
```

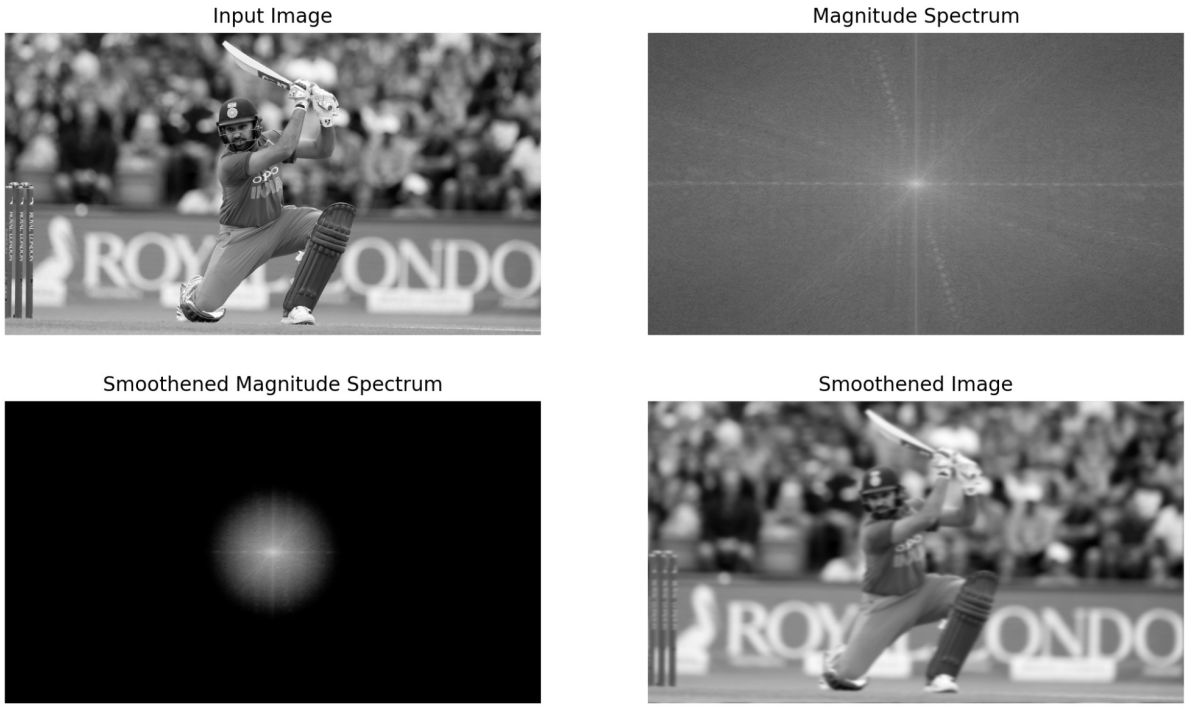


This is the output of applying the ideal low pass filter on the input image. In the third image we can see only the low frequency components are retained. Although the fourth image is the smoothened version of the input image, it has the ringing problem because we have abruptly cut off the high frequency components.

In order to solve this ringing problem, we can use the filters that transition smoothly from the low frequency and high frequency components like the Gaussian Low Pass Filter.

```
def gaussian_low_pass_filter(shifted_ft_image, size):
    rows, cols = shifted_ft_image.shape
    c_row, c_col = rows // 2, cols // 2

    u, v = np.meshgrid(np.arange(rows), np.arange(cols),
indexing='ij')
    mask = np.exp((-D2(u, v, c_row, c_col)) / (2 * size**2))
```

```
    return shifted_ft_image * mask
```



We can see that the output (fourth image) after applying the gaussian low pass filter is smoothened and doesn't have the ringing effect as in the case of the ideal low pass filter.

Sharpening Filter (High Pass Filter)

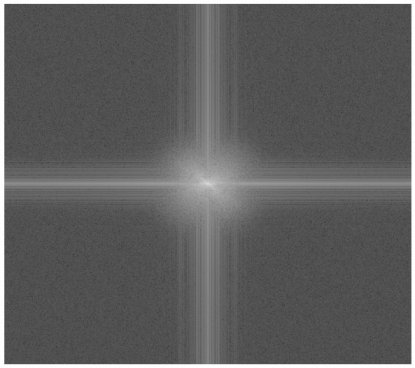
In High Pass Filtering, we mask out the low frequency components in the center and retain the high frequency components. All the Fourier Transform stuff apply here as well.

```
def ideal_high_pass_filter(shifted_ft_image, size):
    rows, cols = shifted_ft_image.shape
    c_row, c_col = rows // 2, cols // 2

    mask = np.ones(shifted_ft_image.shape)
    u, v = np.meshgrid(np.arange(rows), np.arange(cols),
indexing='ij')
    mask[D(u, v, c_row, c_col) <= size] = 0
```

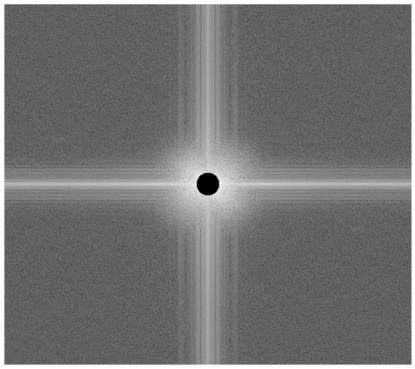
```
    return shifted_ft_image * mask
```

Input Image

Magnitude Spectrum



Sharpened Magnitude Spectrum



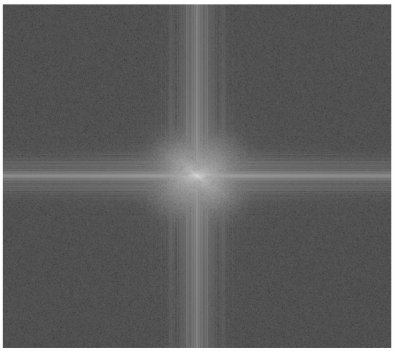
Sharpened Image

Similar to the Ideal Low Pass Filter, the Ideal High Pass Filter produces the ringing effect which can be seen in the fourth image.

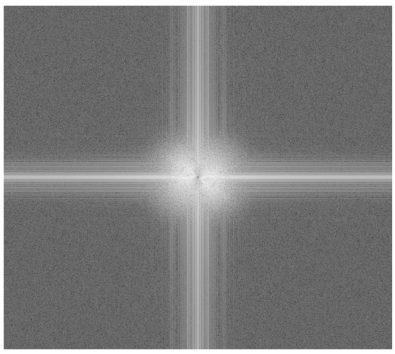
```
def gaussian_high_pass_filter(shifted_ft_image, size):
    rows, cols = shifted_ft_image.shape
    c_row, c_col = rows // 2, cols // 2

    u, v = np.meshgrid(np.arange(rows), np.arange(cols),
indexing='ij')
    mask = 1 - np.exp((-D2(u, v, c_row, c_col)) / (2 * size**2))

    return shifted_ft_image * mask
```

Input Image

Magnitude Spectrum



Sharpened Magnitude Spectrum



Sharpened Image

The Gaussian High Pass Filter solves the ringing problem and the results are shown in the fourth image. With the help of that, we can do the weighted sum with the original image to get the sharpened image exactly like we did with the Laplacian Transform in the spatial domain.