Code For Filtering with Butterworth Notch Filter

```
# take input
img_input = cv2.imread('./two_noise.jpeg', 0)
cv2.imshow('Input', img_input)
img = dpc(img_input)
image_size = img.shape[0] * img.shape[1]
# fourier transform
ft = np.fft.fft2(img)
ft_shift = np.fft.fftshift(ft)

magnitude_spectrum_ac = np.abs(ft_shift)
magnitude_spectrum = 20 * np.log(np.abs(ft_shift))
magnitude_spectrum_scaled = min_max_normalize(magnitude_spectrum)
```

```
def ButterworthFilter(img, point_list, D, n):
    M = img.shape[0]
    N = img.shape[1]
    H = np.ones_like(img.shape, np.float32)
    for uk, vk in point_list:
        for u in range (M):
            for v in range (N):
                Dk = (u - M // 2 - uk) ** 2 + (v - N // 2 - vk) ** 2
                Dk = math.sqrt(Dk)
                D_k = (u - M // 2 + uk) ** 2 + (v - N // 2 + vk) ** 2
                D_k = math.sqrt(D_k)
                if(Dk == 0.0 or D_k == 0.0):
                    H[u, v] = 0.0
                    continue
                H[u, v] *= (1 / (1 + (D / Dk)) ** (2 * n)) * (1 / (1 + (D / D_k)) ** (2 * n))

    return H
```

```
Final = magnitude_spectrum_ac * H

ang = np.angle(ft_shift)
final_result = np.multiply(Final, np.exp(1j * ang))


final_result_back = np.real(np.fft.ifft2(np.fft.ifftshift(final_result)))
final_result_back_scaled = min_max_normalize(final_result_back)

cv2.imshow("Final Output",final_result_back_scaled)
cv2.imwrite("Final_Output.jpeg",final_result_back_scaled)
```

Fig 4.4: Input Image containing two noise

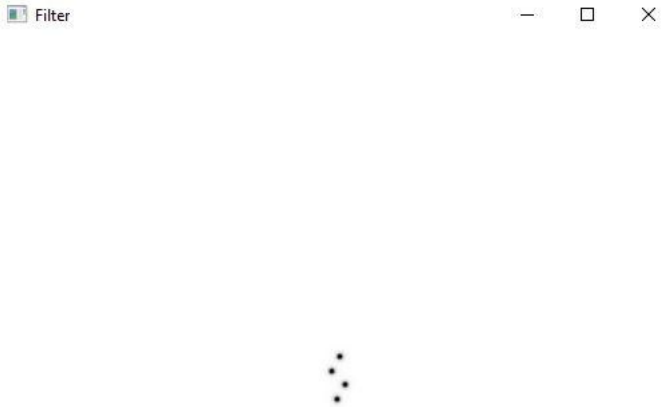

Fig 4.5: Butterworth Notch Filter

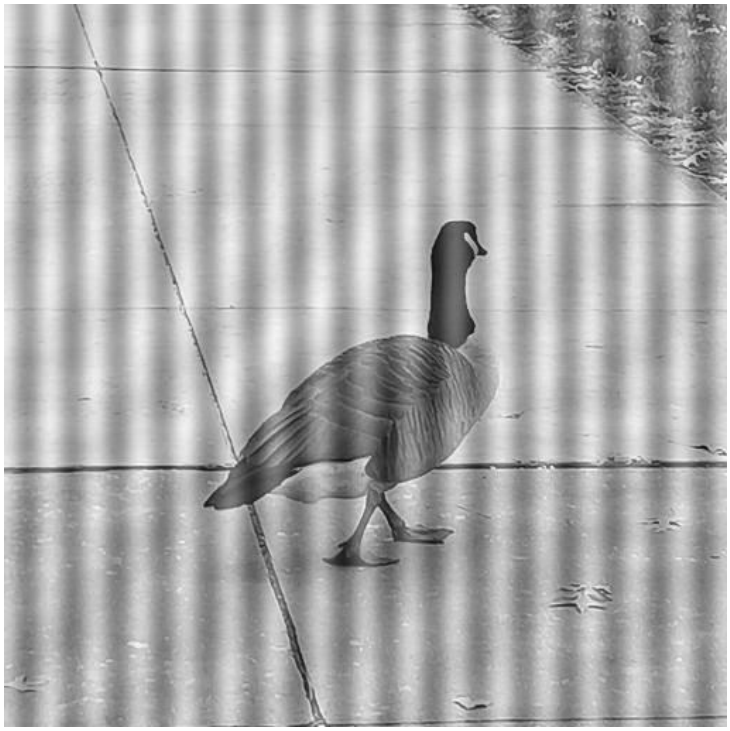

Fig 4.6: Output Image

Code For Homomorphic Filtering

```
def IlluminationPatternGenerator(img, x, y, sigma):
    pattern = np.zeros_like(img, np.float32)
    for i in range (x):
        for j in range (y):
            constant = 2 * math.pi * (sigma ** 2)
            result = -(i * i + j * j)
            result /= 2 * (sigma ** 2)
            numerator = math.exp(result)
            pattern[i, j] = numerator / constant
    return pattern

pattern1 = IlluminationPatternGenerator(img, 400, 400, 100)
pattern2 = IlluminationPatternGenerator(img, 500, 500, 85)

pattern2 = np.flip(pattern2, 0)
pattern2 = np.flip(pattern2, 1)

pattern = pattern1 + pattern2
pattern = cv.normalize(pattern, None, 0, 255, cv.NORM_MINMAX)
pattern = pattern.astype(np.uint8)
```

```
def getHomomorphicKernel():
    gh = 1.4
    gl = 0.5
    D0 = 8
    c = 3
    M = img.shape[0]
    N = img.shape[1]
    kernel = np.zeros(img.shape)
    for i in range (M):
        for j in range (N):
            dk = np.sqrt((i - M // 2) ** 2 + (j - N // 2) ** 2)
            power = -c * ((dk ** 2) / (D0 ** 2))
            kernel[i, j] = (gh - gl) * (1 - np.exp(power)) + gl
    return kernel


kernel = getHomomorphicKernel()

ft = np.fft.fft2(corrupted_img)
ft_shift = np.fft.fftshift(ft)

magnitude_spectrum_ac = np.abs(ft_shift)
magnitude_spectrum = 20 * np.log(np.abs(ft_shift))
magnitude_spectrum_scaled = min_max_normalize(magnitude_spectrum)

cv.imshow("Magnitude Spectrum", magnitude_spectrum_scaled)
cv.imwrite("./Magnitude_Spectrum.jpg", magnitude_spectrum_scaled)

ang = np.angle(ft_shift)
output = np.multiply(magnitude_spectrum_ac * kernel, np.exp(1j * ang))

img_back = np.real(np.fft.ifft2(np.fft.ifftshift(output)))
img_back_scaled = min_max_normalize(img_back)
```

Fig 4.7: Input Image

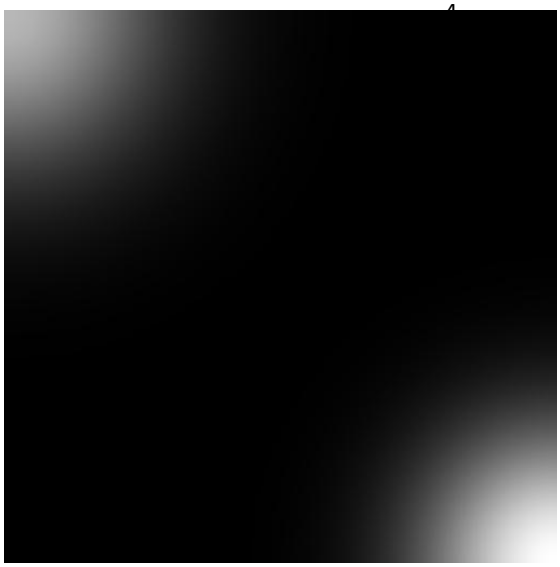

Fig 4.8: Illumination Pattern

Fig 4.9: Corrupted Image

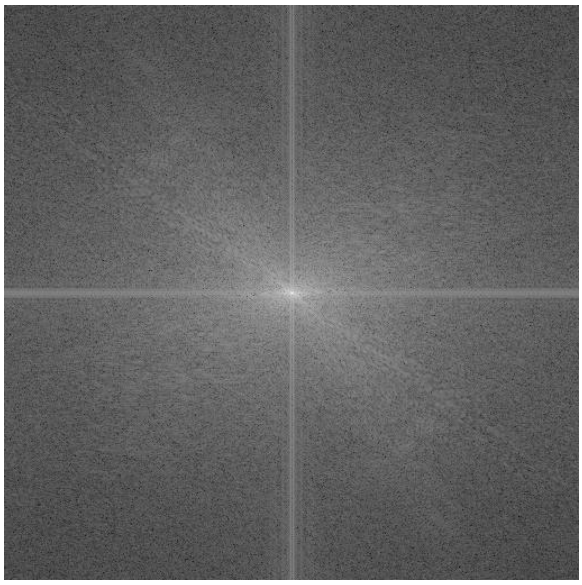

Fig 4.10: Frequency Spectrum of Corrupted Image

Fig 4.11: Output Image