Image Processing

HW3

Lian Natour 207300443

Mohammad Mhneha 315649814

Problem 1:

Image 1:
In this image, we applied a filter that averages each row using a custom kernel with 1s in the middle row and 0s elsewhere, normalized to ensure accurate averaging. The given image looks uniform across each row, with all pixels in the same row having the same intensity, indicating this type of averaging operation. The MSE we got between the given image and the recreated image is 0.05.

Given image : recreated image:

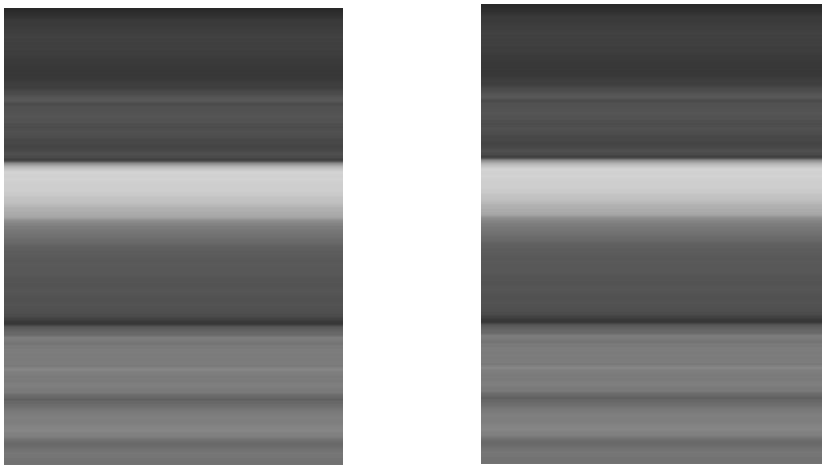


Image 2:
In this image, we applied a Gaussian Blur filter with a kernel size of (11, 11) and sigma 15, along with borderType=cv2.BORDER_WRAP, to smooth the details. The given image looks blurred, with reduced sharpness and softened edges, which is indicative of Gaussian Blur. The MSE we got between the given image and the recreated image is 0.1956.

Given image : recreated image:

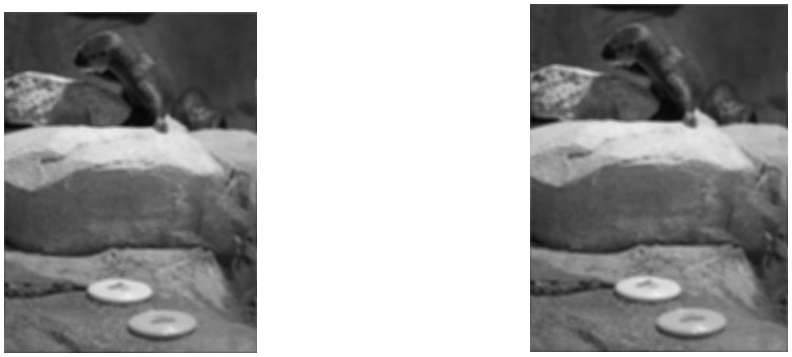


Image 3:
In this image, we applied a Median Filter with a kernel size of 11 to reduce noise while preserving edges. We chose the Median Filter because the given image retains smoothness while keeping structural details, which aligns with the properties of this filter. The MSE between the given and recreated image is 0.3676.

Given image :

recreated image:

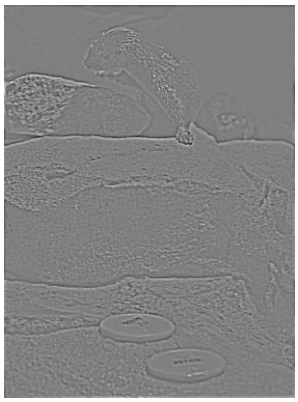
Image 4:
The image exhibits a vertical blur effect, resembling duplication along the y-axis. To replicate this, we applied a vertical averaging kernel with 15 rows (1/15 for each value. The final MSE was 0.31, indicating a close match to the given image.

Given image :

recreated image:

Image 5:
In this image we applied the Bsharp formula taught in the lecture: Bsharp=B−BBlurB , where BBlurB is a Gaussian-blurred version of the original image B. We enhanced the result by adding a constant num to adjust brightness and intensity levels. Through iterative optimization, we identified the best parameters: kernel size (11,11), sigma 5.95, and num=128 , which minimized the MSE to 3.10.

Given image :



recreated image:

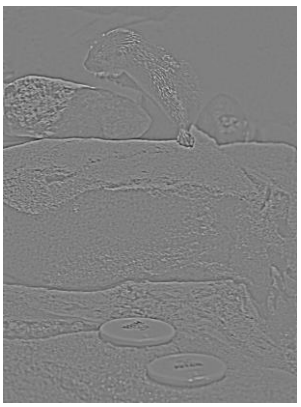
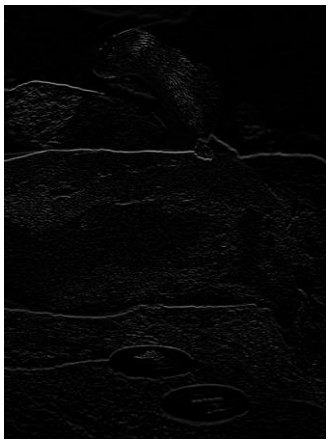


Image 6:
For Image 6, we applied edge detection using a custom kernel aligned with the y-axis. The kernel: [[-0.34, -0.34, -0.34], [0.0, 0.0, 0.0],[0.34, 0.34, 0.34]].
This kernel enhances vertical transitions by subtracting upper and lower pixel intensities. Using OpenCV's filter2D, we applied this filter to extract vertical details, achieving a low MSE of 4.926.

Given image :



recreated image:

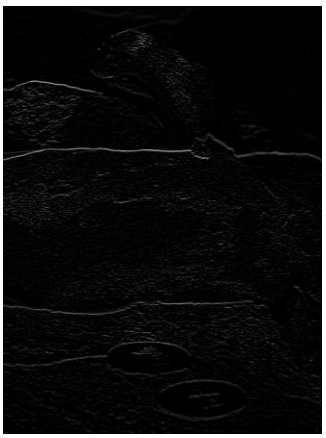


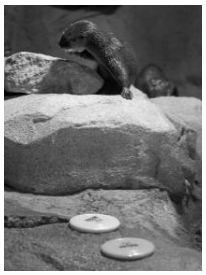
Image 7:
the image was shifted vertically in a cyclic manner, so we created a vertical kernel with a single value set at the first position to simulate this cyclic shift. This kernel, when applied using filter2D with BORDER_WRAP, effectively shifts the image up by half its height while wrapping the pixels around. Using this approach, we achieved an MSE of 1.18.

Given image :

recreated image:

Image 8:
For Image 8, we applied the identity kernel, which keeps the image unchanged. This was sufficient because the target image (image_8.jpg) is essentially a grayscale version of the original image with minimal differences. we achieved MSE of 0.3464.

Given image :



recreated image:

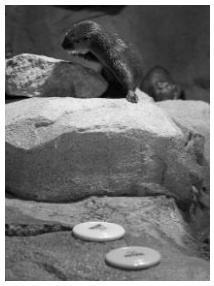
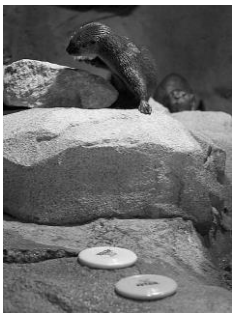


Image 9:
For Image 9,Observing the target image, it appeared sharper and darker compared to the original. Based on this observation, a sharpening filter was chosen, which enhances edges and increases contrast in the image.The sharpening kernel used was: [[-0.5, -0.5, -0.5],[-0.5, 6.0, 0.5], [-0.5, -0.5, -0.5]] / 2. the processed image achieved an MSE of 7.34 compared to the target.

Given image :

recreated image:



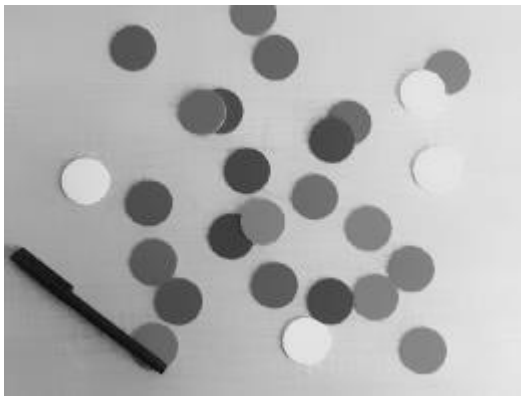
Problem 2:

a) We implemented a bilateral filter to clean Gaussian noise while preserving edges. The filter combines spatial and intensity differences to calculate weights for smoothing. To handle boundary pixels, the input image was padded using the reflect mode. A Gaussian kernel was precomputed for spatial distances, and for each pixel, intensity-based Gaussian weights were calculated. The combined weights were normalized, and the filtered value was computed as the weighted sum of neighboring pixels.
here is the code of the process as required:

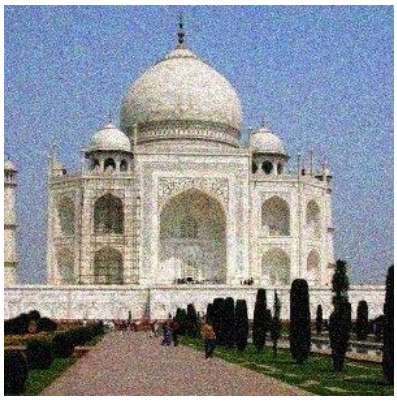
```
# Create spatial Gaussian kernel
x, y = np.meshgrid( *xi: np.arange(-radius, radius + 1), np.arange(-radius, radius + 1))
gs = np.exp(-(x ** 2 + y ** 2) / (2 * stdSpatial ** 2))

# Iterate over each pixel in the original image
for i in range(radius, padded_height - radius):
    for j in range(radius, padded_width - radius):
        # Extract the window centered around the pixel
        window = padded_image[i - radius:i + radius + 1, j - radius:j + radius + 1]

        # Compute intensity Gaussian
        gi = np.exp(-((window - padded_image[i, j]) ** 2) / (2 * stdIntensity ** 2))

        # Compute the bilateral filter response
        weights = gs * gi
        weights /= weights.sum()
        cleanIm[i - radius, j - radius] = np.sum(weights * window)
```

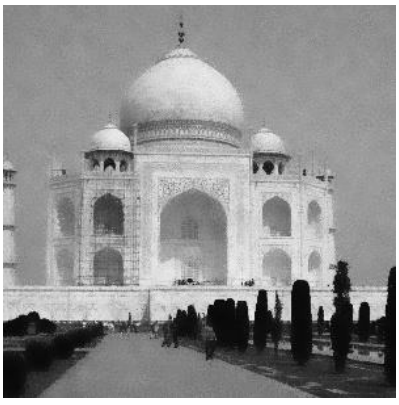
b) In this section, we applied the custom bilateral filtering function implemented in Section A to clean three provided noisy images: taj.jpg, NoisyGrayImage.png, and balls.jpg. Bilateral filtering is highly effective in reducing noise while preserving edges.

Here's an explanation of my approach:

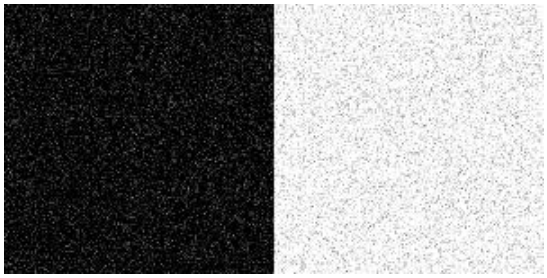


This image primarily suffered from low resolution, with visible compression artifacts. The goal was to smooth the image to reduce the artifacts while preserving the round shapes of the balls. The area around the pen in the image posed a challenge, as smoothing there risked affecting the clarity of the balls. After experimenting, we used a radius of 9, a spatial standard deviation of 10, and an intensity standard deviation of 10. This combination provided a balance between smoothing the artifacts and maintaining object clarity.





This image contained noticeable noise but also had intricate details, particularly around the edges of the Taj Mahal structure. To clean the noise effectively while maintaining these details, we used a radius of 5, a spatial standard deviation of 50, and an intensity standard deviation of 30. These parameters allowed the bilateral filter to preserve edges while smoothing out the noise across the flat regions of the image.



The white side of this image exhibited heavy noise in the form of dense black dots, creating a challenge to clean without overly blurring the image. While the bilateral filter cleaned the majority of the noise, a faint blurring of the central transition line between black and white remained visible. After testing several parameter combinations, we chose a radius of 7, a spatial standard deviation of 7, and an intensity standard deviation of 100.

Problem 3:
a)


To address the noise in the given image "broken.jpg," a combination of median blurring and bilateral filtering was applied to achieve effective noise reduction while preserving important details such as edges. The bilateral filter function, implemented earlier, was specifically designed for this purpose as it allows for edge-preserving smoothing.

Initially, a median blur with a kernel size of 5 was applied as a preprocessing step to reduce salt-and-pepper noise. Following this, the bilateral filter that we implemented was used with carefully selected parameters: a radius of 5, a spatial standard deviation of 75, and an intensity standard deviation of 75.

b) we used the "noised_images.npy" file, which contains 100 noisy versions of the same image, to generate a cleaned output. By averaging these images.To optimize the result, we applied a multi-stage filtering process to each noisy image. First, we used median blur (kernel size 5) to remove salt-and-pepper noise. Next, bilateral filtering (d=7, sigmaColor=25, sigmaSpace=25) was applied to smooth Gaussian noise while preserving edges. These parameters were chosen after experimenting with multiple values to balance noise reduction and edge preservation. Finally, the processed images were averaged to enhance consistent details further and suppress residual noise.

Here is the image we got :