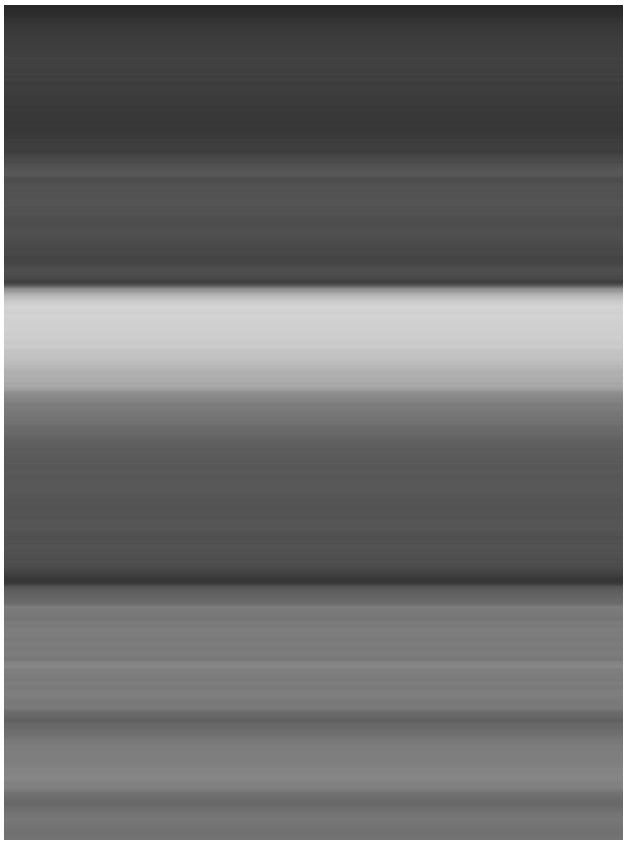
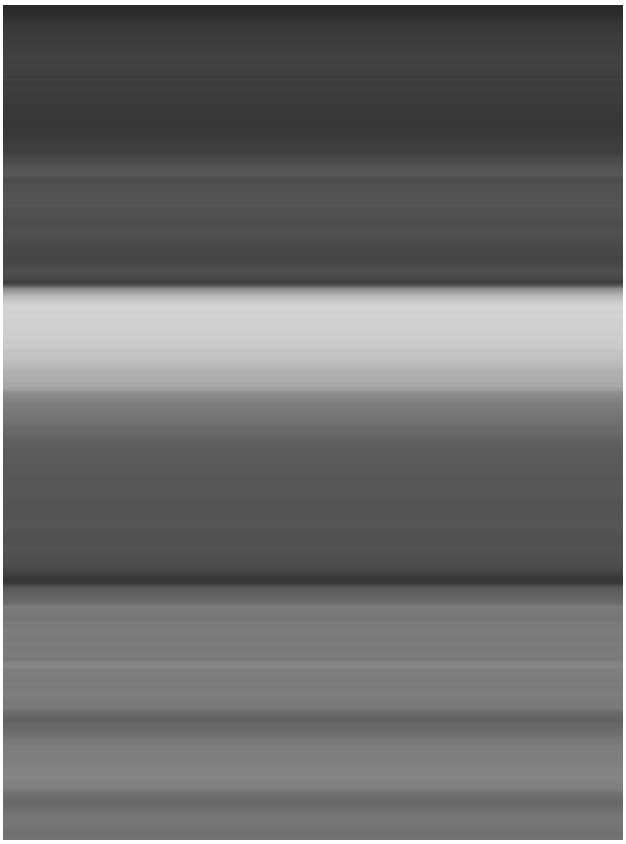
Image Processing: Assignment #3

Problem 1 – What happened here

Image 1

given **our recreated** (bonus)





This image was created with convolution. The mask calculates the mean of the gray values in each row and sets all pixels of this row the calculated mean value. Near the boundary wrap around method Is used. We chose this because it appears that each row has a constant gray value for its entire length and different rows has a different gray value, so the average is by row.

Bonus:

The kernel is of size 1X592 (1 row with image width columns) each cell in kernel has the same value $\frac{1}{\text{image\_width}} = \frac{1}{592}$. Convolution between this kernel and the image with wrap around method near the border will give to each pixel the mean gray value of the row. image_1 MSE is: 0.0.

Image 2

given **our recreated** (bonus)

This image was created with convolution with averaging kernel. Near the boundary wrap around method Is used. We chose this because the picture is blurred, this is not median blurring because if we look at the back of the otter, there is no clear border between its back and the background of the image, which is better preserved in gaussian and median blurs. Wrap around method in the image border can be seen for example on the top and bottom of the image, the bottom has some darker lines which came from the dark pixels on the top of the image, and vice versa, the top border has some brighter lines from the bottom.

Bonus:

The kernel is in shape of 11x11 each cell of the kernel has the same value $\frac{1}{11^2}$. Convolution between this kernel and the image with wrap around method near the border will give to each pixel the mean gray value of all the pixels around it with radius 5 and itself. image_2 MSE is: 0.22575865339023235

Image 3

given | **our recreated** (bonus)

This image was created with median blur. We chose this because the picture is blurred, but it preserves edges well, the borders in the image such as around the otter or the rocks. But fine details in the image are not well preserved, such as the eyes of the otter in the back are barely seen, and for the sitting otter, the fingers look like one object.

Bonus:

We retreated the image using cv2.medianBlur, the area size around each pixel to calculate the median value is 11. image_3 MSE is: 0.0

Image 4

given | **our recreated** (bonus)

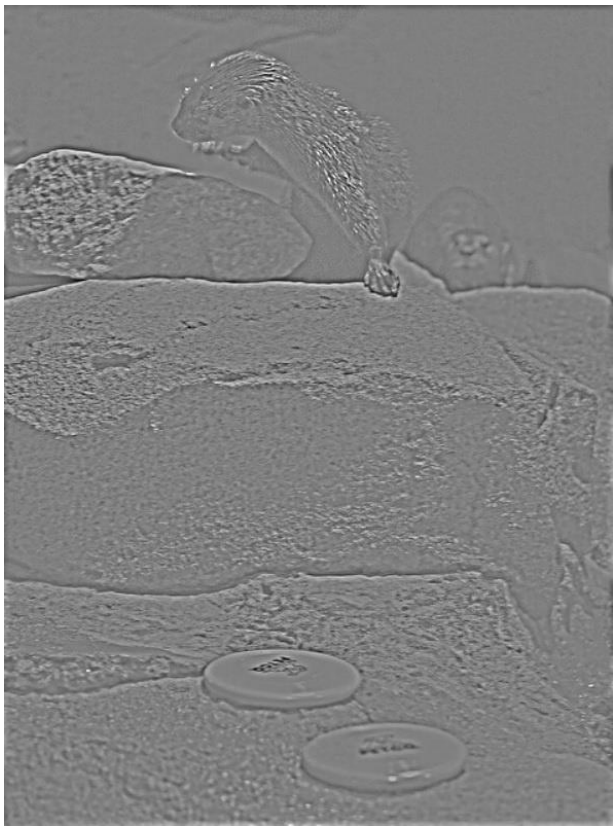
This image was created with convolution with averaging kernel oriented on Y-axis. This image looks like it has a motion blurring on Y-axis. For example, the otter's beard is longer than in the original image while in X-axis there is no blurring in the otter's beard. Wrap around method in the image border is used, it can be seen for example on the top and bottom of the image, the bottom has some darker lines which came from the dark pixels on the top of the image, and vice versa, the top border has some brighter lines from the bottom.

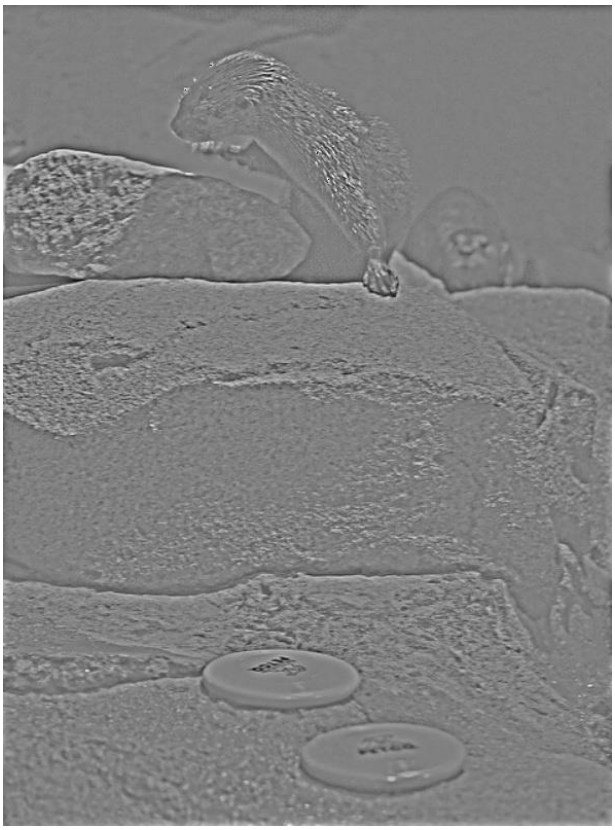
Bonus:

We built an average kernel with size of 15X1. 15 rows with 1 column. each cell of the kernel has the same value $\frac{1}{15}$. 1 column is because of Y-Axis orientation, and 15 rows, we tried a few numbers of rows until we got MSE of 0. The Convolution between this kernel and the image with wrap around method near the border will give to each pixel the mean gray value of all the pixels above and below it with radius 7 and itself. image_4 MSE is: 0.0

Image 5

given | **our recreated** (bonus)





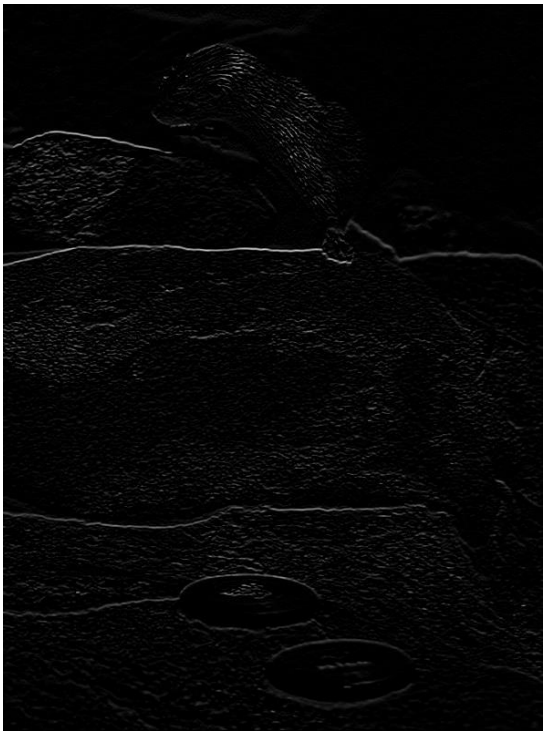
This image was recreated with part of the method to sharpening image.
1. Calculate the blurred image with Gaussian kernel.
2. Subtract the blurred image from the image that should be enhanced.
After these 2 steps, to get the given image, it should be clipping to the range [0, 255].
We choose it because it looks like the sharp image in the sharpening method before the addition to the image that should be enhanced. It contains all edges and fine details (high frequency) of the enhancing image. Wrap around method in the image border is used, it can be seen for example on the right border, pixels which calculated with pixels near the left border for example the top rock in the left side cause some black artifact on the right-side border.

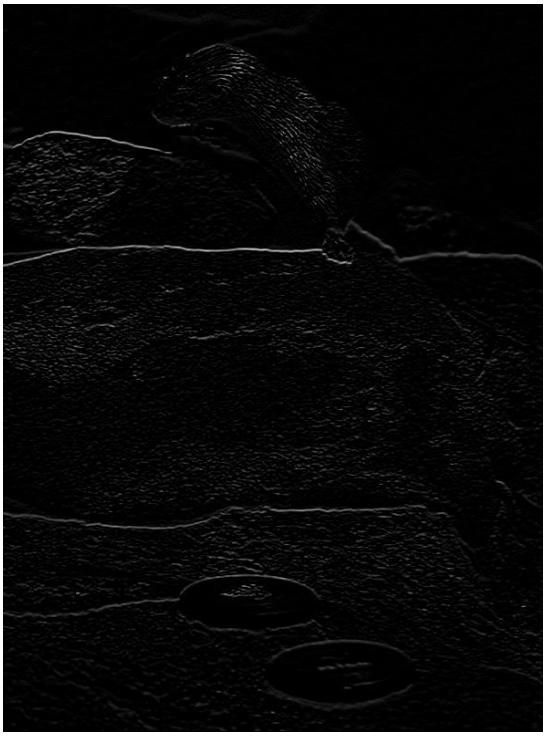
Bonus:

We first calculate blurred image with gaussian filter with filter size 19, sigma 4 and wrap method on borders. Then we Subtract the blurred image from the original image and clipped it to the range [0,255] by adding 127 to all pixels. image_5 MSE is: 3.324174033055612.

Image 6

given **our recreated** (bonus)





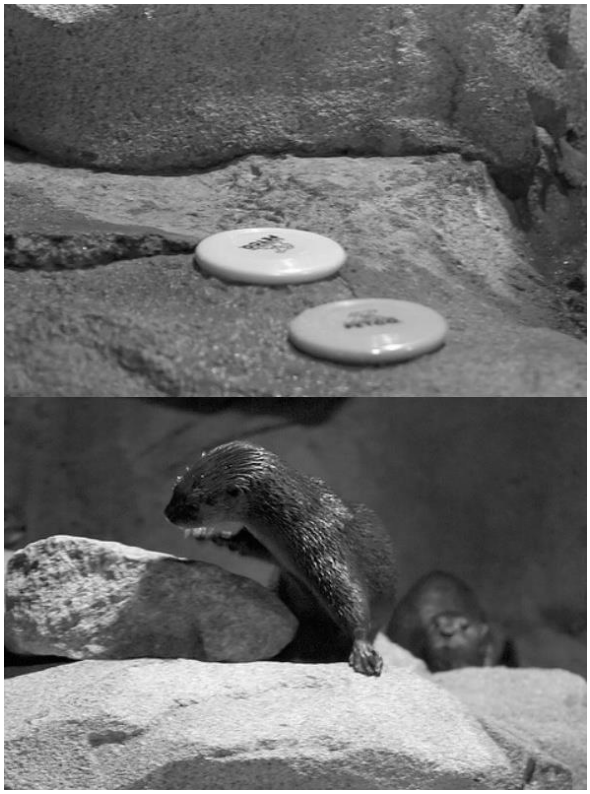
This image was created with convolution with Laplacian kernel for horizontal edges. We chose it because all horizontal edges can be strongly seen compared to the vertical edges. For example, all horizontal edges of the rocks seen while vertical not.

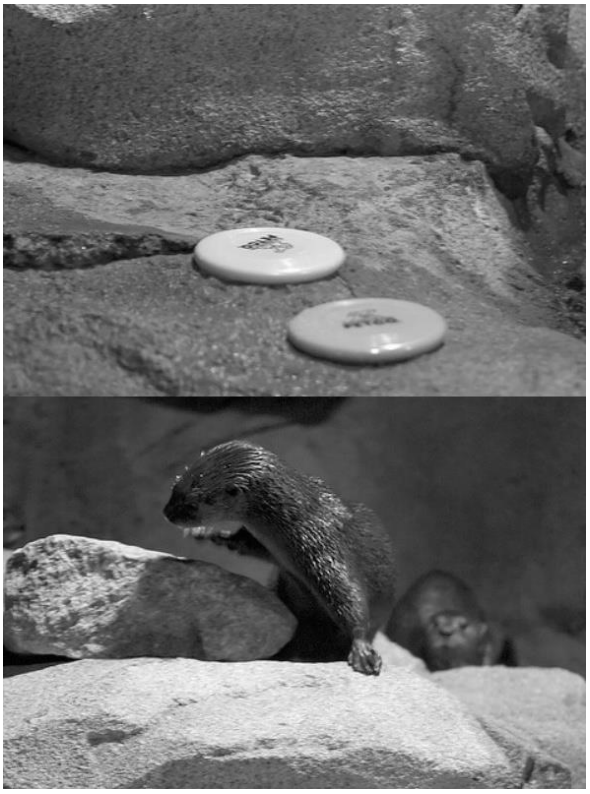
Bonus:

We use kernel $\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$ for horizontal edge detection, use of narrow kernel (1 column) for detect horizontal edged, is less susceptible to vertical variations. Then we perform convolution with this kernel, and on boundary we use wrap method. image_6 MSE is: 0.0

Image 7

given **our recreated** (bonus)





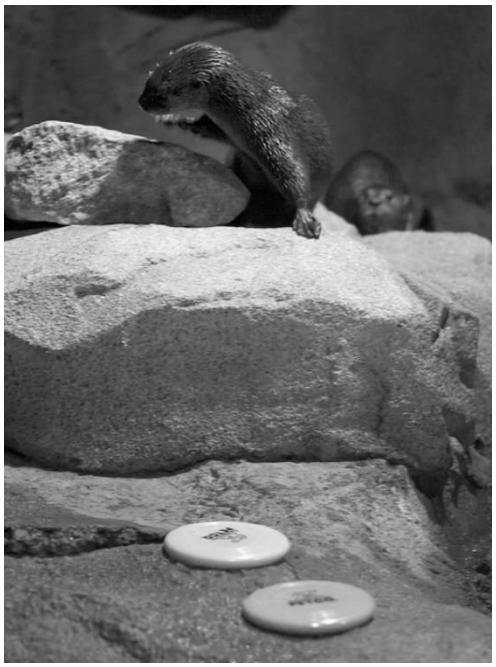
This image was created with convolution that moves the whole image down/up cyclically $\frac{\text{image hight}}{2}$ pixels. We chose it because it seen in the picture that the image Is split in a half exactly (exactness was tested in paint software) and the upper half is now in the bottom of the image and vice versa. And for the cyclically we need to handle the border with wrap method. The kernel size must be in the shape of the image but with 1 more row, to gain that the number of pixels to move is half of the image when the origin in the kernel is in the center. The whole kernel is with 0 values except the value in the cell in the last row (image Hight +1) and the middle column ($\frac{width}{2}$), this is because we want to move only on Y-axis.

Bonus:

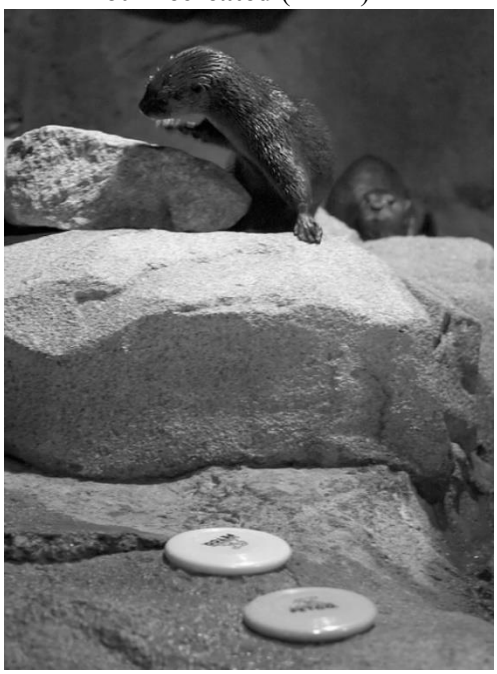
We build the exact kernel that is mentions above and run convolution with wrap method on borders. image_7 MSE is: 0.0.

Image 8

given



our recreated (bonus)



This image is the same image as the original but in gray scale.
We check the mse between the given image and the original image after reading it in gray scale. They have MSE = 0.

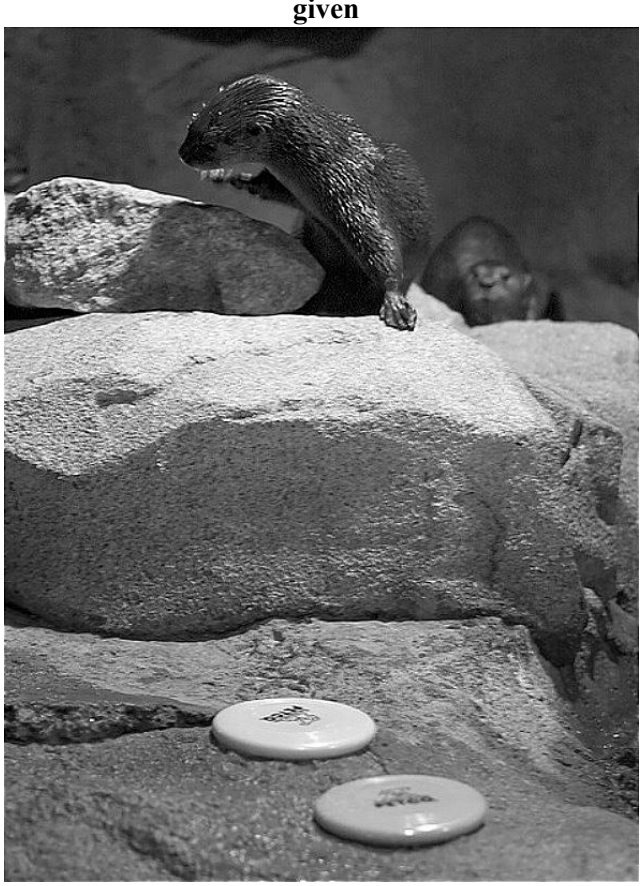
Bonus:

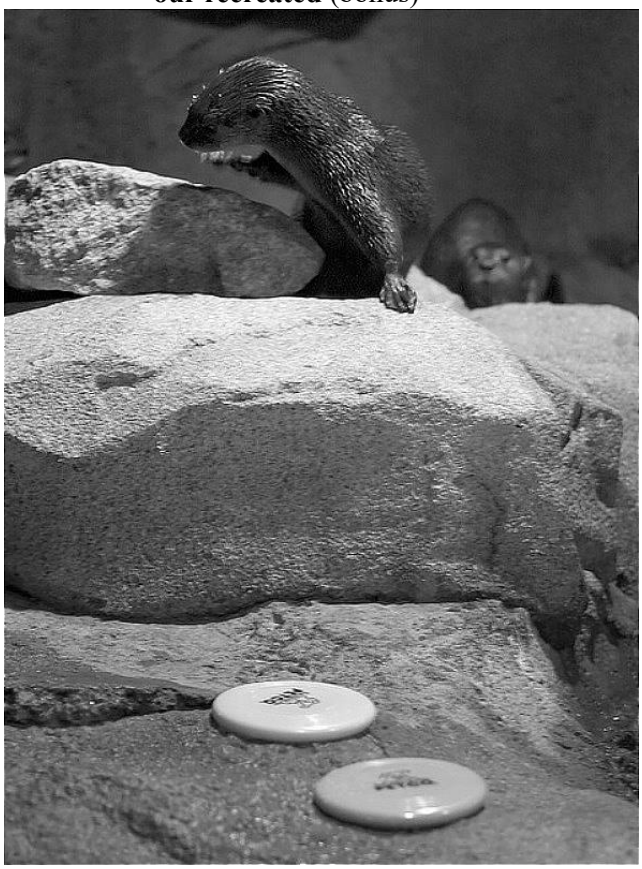
As much as it the same image in gray scale, we didn't perform anything except read it in gray scale.

image_8 MSE is: 0.0

Image 9

given | **our recreated** (bonus)





This image was created with a sharpening filter. It can be seen in the image that edged (for example the rock edges are clearer than in the original image in gray scale) and fine details (such as the otter's fur) looks enhanced. They look clearer than in the original image in gray scale. Wrap around method in the image border is used, it can be seen for example on the right border, pixels which calculated with pixels near the left border for example the top rock in the left side cause some black artifact on the right-side border.

Bonus:

We build the kernel as follows:

$$G = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 4 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \frac{1}{12} \rightarrow \delta - G = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} * \frac{1}{12} \rightarrow S(1) = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 20 & -1 \\ -1 & -1 & -1 \end{bmatrix} * \frac{1}{12}$$

We tried different types of blurring masks. This G gave us the smaller MSE. We perform the convolution twice on the image with S (1) and this is giving us this result.
image_9 MSE is: 10.549189697216013

Problem 2 – Biliteral analysis:

a)

Bilateral Filtering Concept:
Bilateral filtering is a technique in image processing that reduces noise while preserving edges. It does this by considering the difference in intensity between neighbouring pixels as well as their spatial distance. The filter uses two Gaussian functions: one for space (spatial proximity) and one for intensity (pixel value similarity).

Key Parts of the Implementation:

- Defining Gaussian Functions:
  We define two Gaussian functions, one for spatial weighting (gs) and one for intensity weighting (gi). The spatial Gaussian (gs) depends only on the distance from the center pixel, while the intensity Gaussian (gi) depends on the difference in pixel intensity.

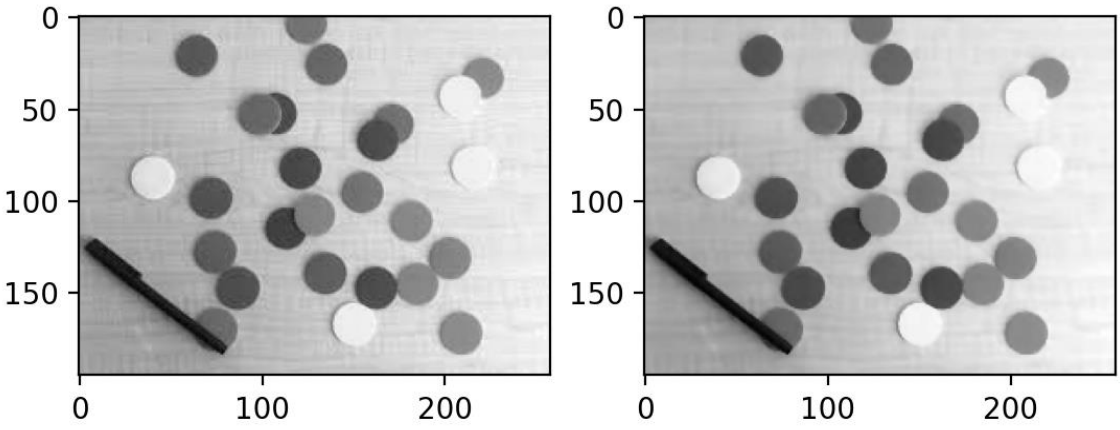
- Creating a Meshgrid for the Window:
  A meshgrid is used to calculate the spatial distance from each pixel within the window to the center pixel. np.meshgrid is a convenient way to generate the coordinate grid needed for the spatial component of the bilateral filter.

```
# Since gs is dependent only on the distance, it can be precomputed once
y, x = np.meshgrid( *xi: np.arange(-radius, radius + 1), np.arange(-radius, radius + 1))
gs = np.exp(-(x ** 2 + y ** 2) / (2 * stdSpatial ** 2))
```

- Applying the Filter:
  We iterate over each pixel in the image, extract the neighbourhood window, calculate the intensity Gaussian, combine it with the spatial Gaussian, normalize the weights, and apply the filter.

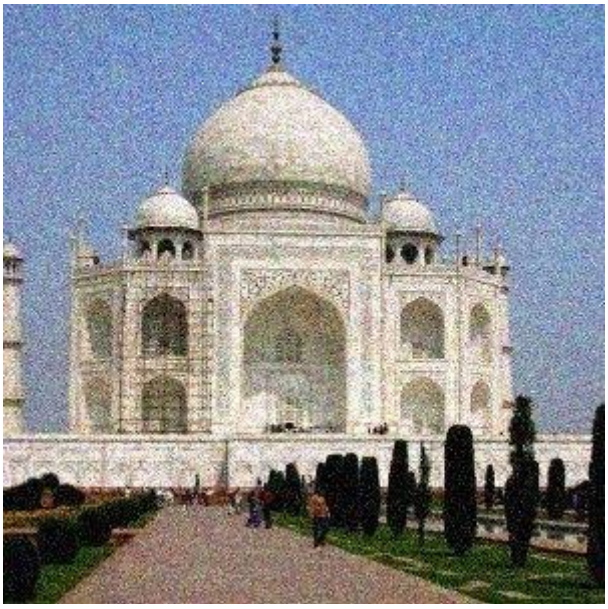
```
# Iterate over each pixel in the image
for i in range(radius, height - radius):
    for j in range(radius, width - radius):
        # Extract the local region (window)
        window = im[i - radius:i + radius + 1, j - radius:j + radius + 1]

        # Calculate the intensity Gaussian function (gi)
        # This depends on the intensity difference of the central pixel and its neighbours
        gi = np.exp(-((window - im[i, j]) ** 2) / (2 * stdIntensity ** 2))

        # Calculate the final mask by multiplying the spatial and intensity functions
        mask = gi * gs

        # Normalize the mask to preserve the average gray level
        mask /= mask.sum()

        # Apply the mask to the window, sum to get the new pixel value
        cleanIm[i, j] = (mask * window).sum()
```

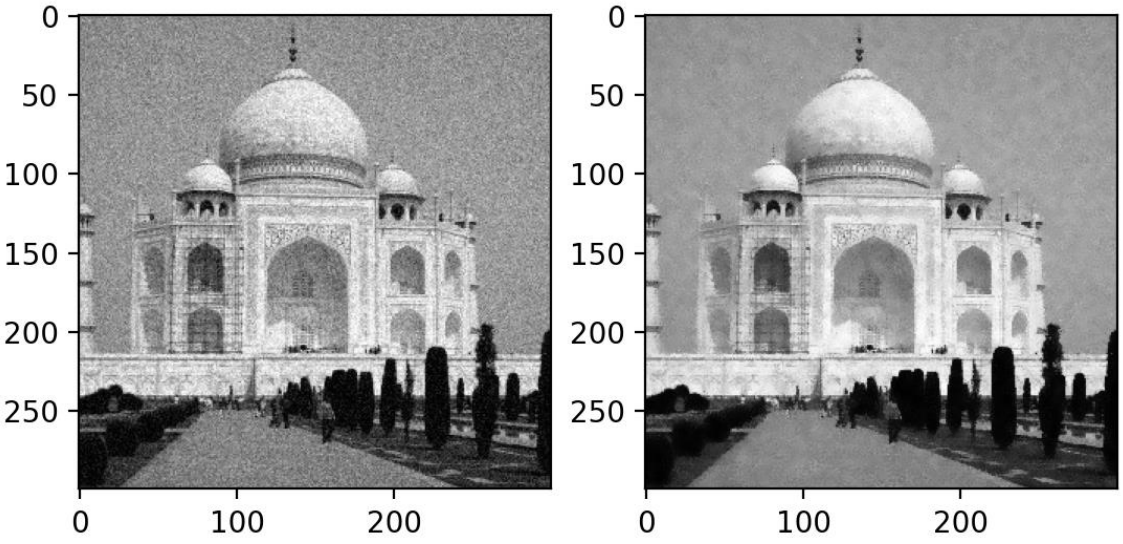
b)




```
clear_image_b = clean_Gaussian_noise_bilateral(image, radius: 1, stdSpatial: 10, stdIntensity: 15)
```

- **Radius (1):** we chose a small radius because compression artifacts like blockiness or mosquito noise often affect small areas. A larger radius might over-smooth the image and erase important details.
- **Standard Deviation for Spatial Weight (10):** a smaller value means that only nearby pixels have a significant weight. We chose a modest value to avoid over-smoothing and to ensure that the filter does not blur the edges of the colored circles and the pen.
- **Standard Deviation for Intensity Weight (15):** This value controls how much the filter considers differences in pixel intensity. A larger value means that more distant intensities will influence the smoothing. We chose a value that's not too high to ensure that the smoothing does not ignore the compression artifacts, which can be subtle in intensity variation.

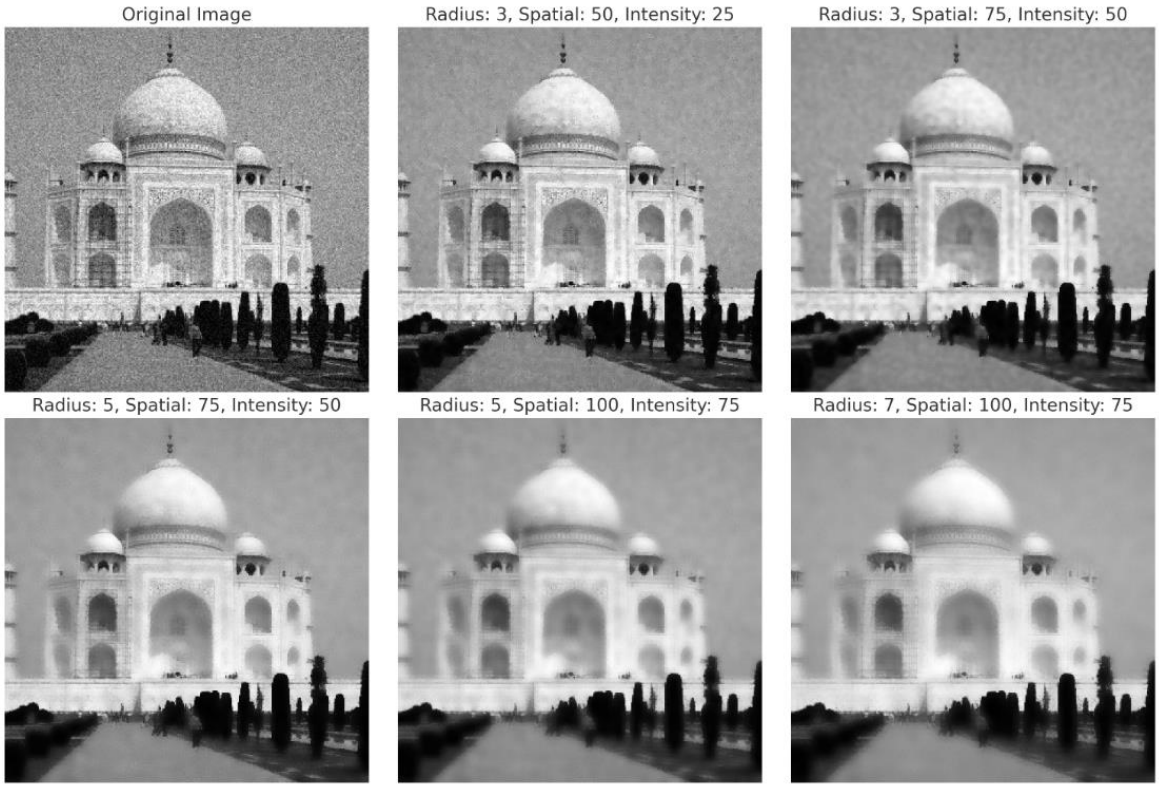


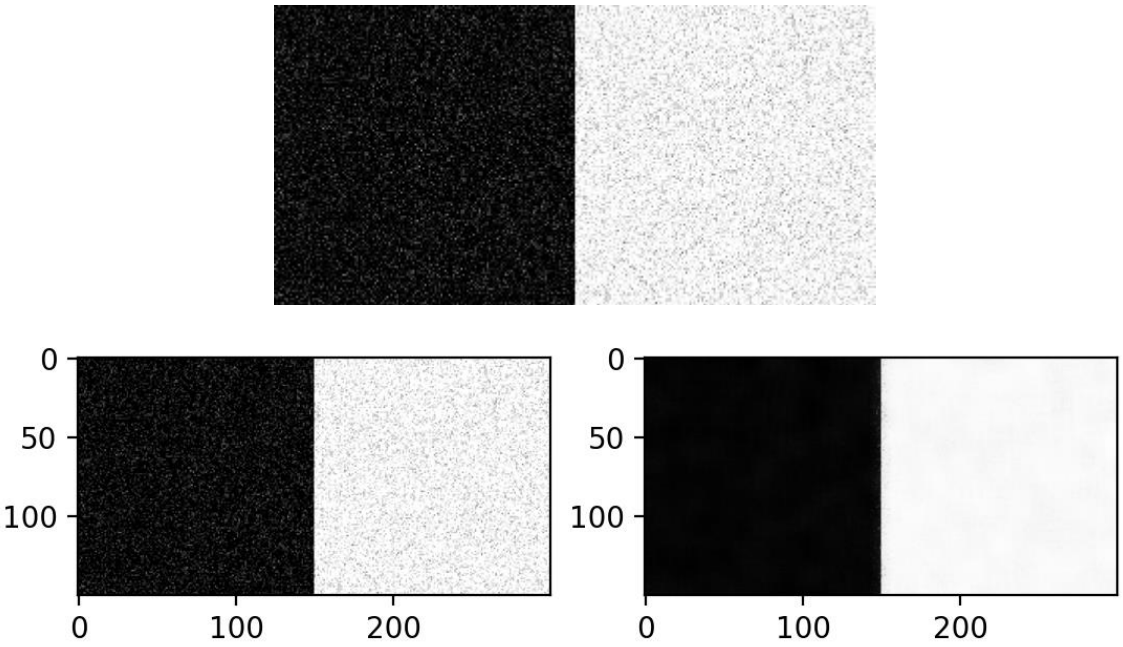



```
clear_image_b = clean_Gaussian_noise_bilateral(image, radius: 3, stdSpatial: 50, stdIntensity: 25)
```

- **Radius (3):** This is a relatively small neighbourhood, which is good for preserving details because it limits the influence of distant pixels, which are less likely to be similar.
- **Standard Deviation for Spatial Weigh (50):** A value of 50 is moderate and suggests that the spatial component of the Gaussian filter will provide a blend of pixels that are not too far from the target pixel, preserving edges while still reducing noise.
- **Standard Deviation for Intensity Weight (25):** This parameter determines how much influence differences in pixel intensity (brightness) have on the filtering. A value of 25 means that only pixels with similar intensities will be averaged together significantly. This prevents the blurring of edges and details because areas of high contrast (which often correspond to edges in the image) will not be averaged out as much.

Testing other params:






```
clear_image_b = clean_Gaussian_noise_bilateral(image, radius: 7, stdSpatial: 50, stdIntensity: 100)
```

- **Radius (7):** This size is large enough to smooth over the noise (which usually affects only a few pixels at a time) but not so large that it would overly blur important details.

- **Standard Deviation for Spatial Weigh (50):** slight variations will be smoothed out, but strong edges with significant intensity differences should remain relatively unaffected.
- **Standard Deviation for Intensity Weight (100):** A value of 100 is quite large, meaning that the spatial proximity will have a significant impact on the smoothing process. This encourages a stronger blending of pixels that are close together, which can be very effective for noise that is not spatially correlated.

Problem 3 – Fix me!

A) In the image it appears that the noise is a mixture of Gaussian noise and "salt and pepper" noise. To improve the image and remove the noise, we applied first biliteral filter and then median filter.

Biliteral Filter:

Firstly, biliteral filter was applied, it helps to reduce the noise while preserving the edges. The bilateral filter is a non-linear, edge-preserving, and noise reduction smoothing filter. In this filter the weights depend not only on Euclidean distance of pixels, but also on the intensity differences.

Parameters:

- d: Diameter of each pixel neighborhood. We used d=10.
- sigmaColor: Value of σ in the color space. The greater the value, the colors farther to each other will start to get mixed. We used sigmaColor=125.
- sigmaSpace: Value of σ in the coordinate space. The greater its value, the further pixels will mix, given that their colors lie within the sigmaColor range. We used sigmaSpace=25.

Code Snippet:

```
# Apply bilateralFilter
bilateralFilter_image = cv2.bilateralFilter(noisy_image, d=10, sigmaColor=125, sigmaSpace=25)
```

Median Filter:

After the bilateral filter, a median filter was applied. The main idea is that for each pixel, a neighbourhood around the pixel is considered, and the median value of this neighbourhood is computed to replace the pixel value. This is particularly effective at removing 'salt and pepper' noise.

Parameters:

- The kernel size, which defines the size of the neighbourhood used to compute the median. A kernel size of 3 means that the median is computed from the 3x3 square neighbourhood of each pixel.

Code Snippet:

```
# Apply median filter
fixed_image = cv2.medianBlur(bilateralFilter_image, ksize: 3)
```

Result:

Noisy Image

Bilateral Filter Image

Median filter Applied after Bilateral Filter

B) To reduce noise, the np.mean function is used. This function calculates the mean along the specified axis, which in our case is axis=0. This means the averaging is done across the stack of images, for each pixel position, it averages that pixel's value across all the images. This helps in cancelling out the random noise which is different in each image while reinforcing the signal which is the same across all images.

```
# Compute the average image from the stack of noised images
average_image = np.mean(noised_images, axis=0)
```

result: