Image Processing - Assignment 3

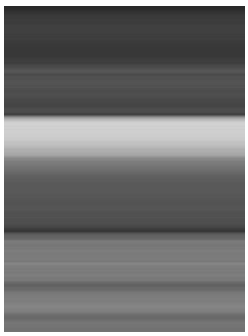
Submitters:

Maya Atwan , ID:314813494

Obaida Khateeb , ID: 201278066

Problem 1 – What happened here

Image 1- operation: Average Filter



From the image we can see the rows with the same color and no discontinuities, this caused by average filter it gives the same avg value for the row and smooth blinding .

Image 2 – operation : Gaussian Blur

The image looks blurry with smooth details . further more we can notice the edges appear lighter than the original image (almost white) . which clearly indicates the use of Gaussian Blur.

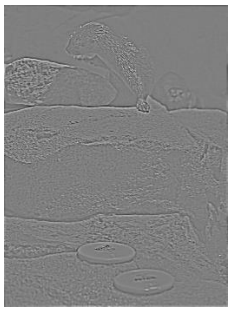
Image 3 – operation : Median Filter

This image appears noise reduced, the details less sharp and smoother , while edges where well preserved this indicates use of median filter.

Image 4 – operation: Average Filter

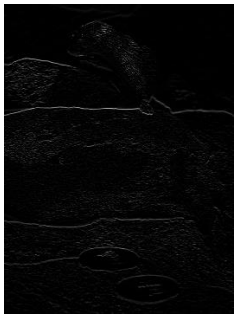
The vertical axis in this image looks smoother and more blended compared with the original image while the horizontal details are preserved. This indicates an average filter applied along the vertical axis.

Image 5-operation: sharpness



Edge sharpness, this is the result of subtracting gaussian blurred version of the image from the original image. (we saw a similar this effect in lecture 5 the little girl image)

Image 6 – operation : Laplacian Filter



The image showcase black with white(bright) horizontal edges which aligning well with Laplacian horizontal edge detection (the same filter effect was applied in the lena image tutorial 5).

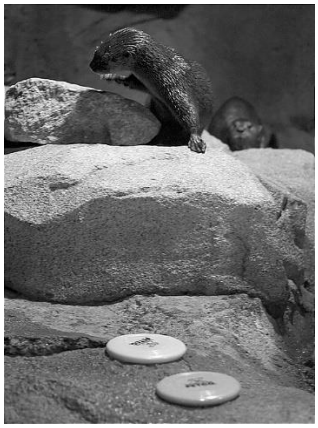
Image 7 – operation : shift in y direction

The image was shifted in y direction , as we can see the upper half of the original image appears at the bottom in this image and the lower half of the original image in now at the top.

Image 8 – operation: no operation

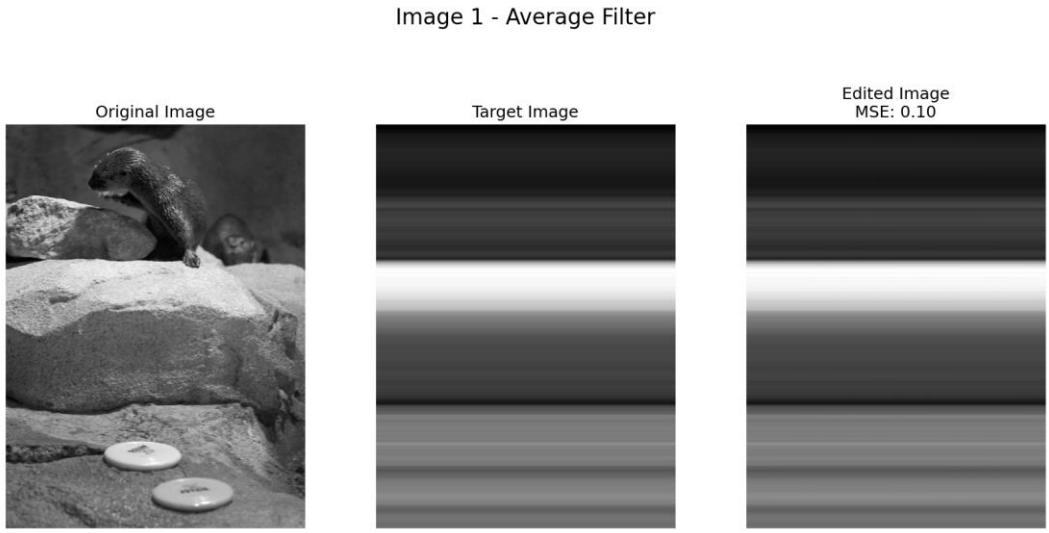
We didn’t notice any spatial operation was applied to the image , we calculate the mse between the 2 images and we got mse=0.34 that’s mean the two images almost identical, the only difference the image has been converted to grayscale.

Image 9 – operation : sharpening



It easy to see that the edges and details are sharper making boundaries more noticeable. That's changing due to sharpening.

Buons:

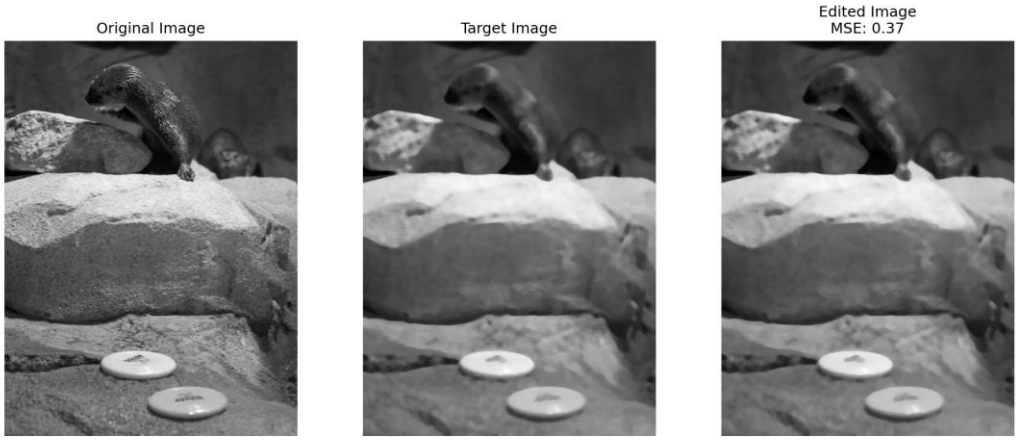


Kernel used: 1*1183 size, in which all the elements are equal to $\frac{1}{1183}$.

Image 2 - Gaussian Blur

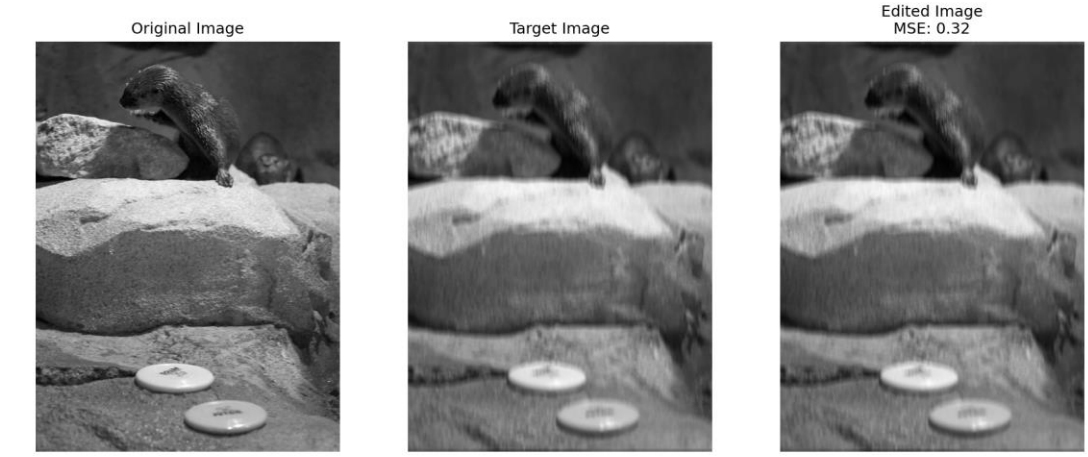
Kernel used: 11*11 size with $\sigma_x = \sigma_y = 11$.

Image 3 - Median Filter



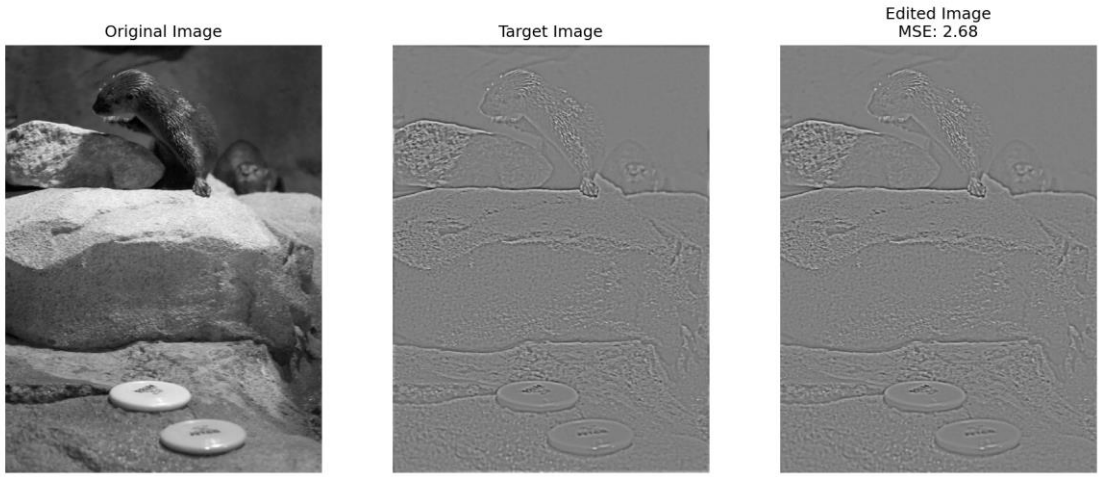
Kernel used: 11*11 size.

Image 4 - Average Filter



Kernel used: 15*1 size, in which all the elements are equal to $\frac{1}{15}$.

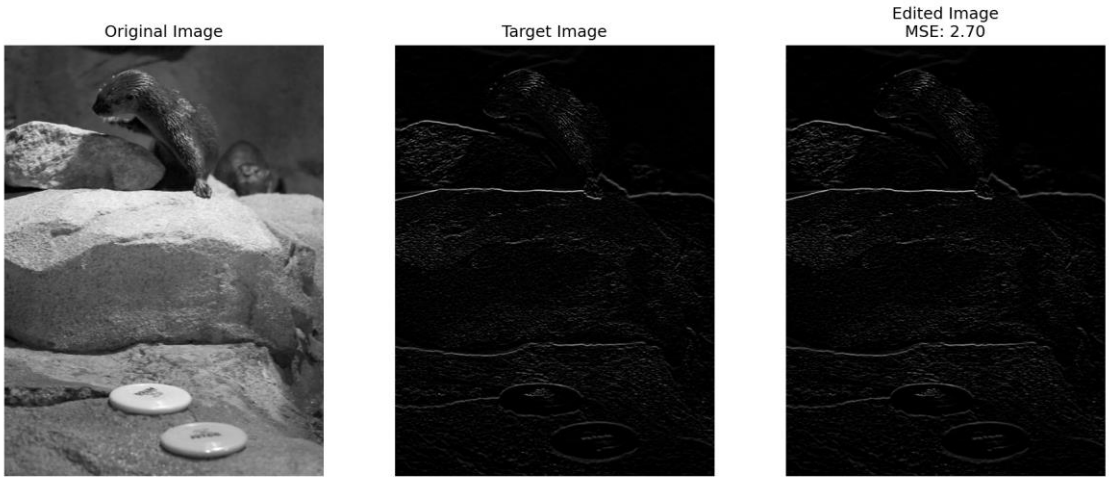
Image 5 - sharpening Filter



Kernel used for blurring the image: 11*11 size with $\sigma = 15$.

Additional operations: substracting the blurred image from the original one, adding 128 to each of the pixels.

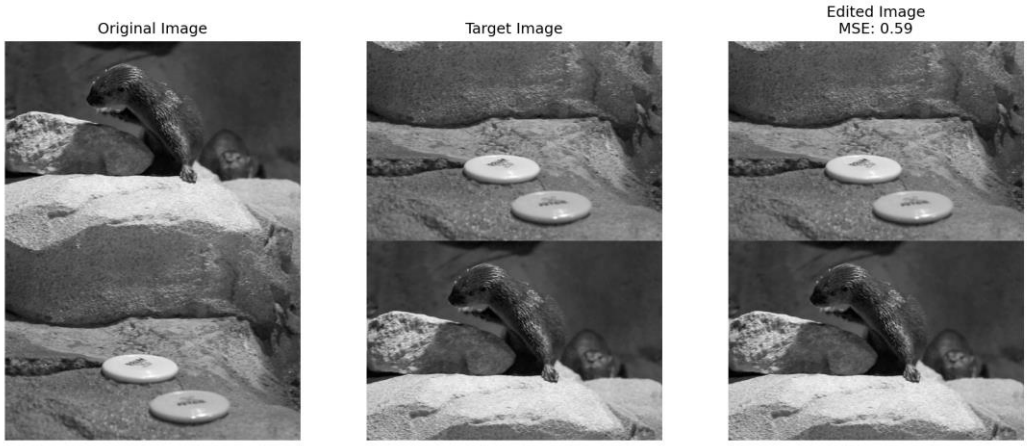
Image 6 - Laplacian Filter



Kernel used:

$$\begin{bmatrix} -0.2 & -0.65239 & -0.2 \\ 0 & 0 & 0 \\ 0.2 & 0.65239 & 0.2 \end{bmatrix}$$

Image 7 - Translation Filter



Kernel used: The translation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 399 \\ 0 & 0 & 1 \end{bmatrix}$$

Image 8 - Original Image

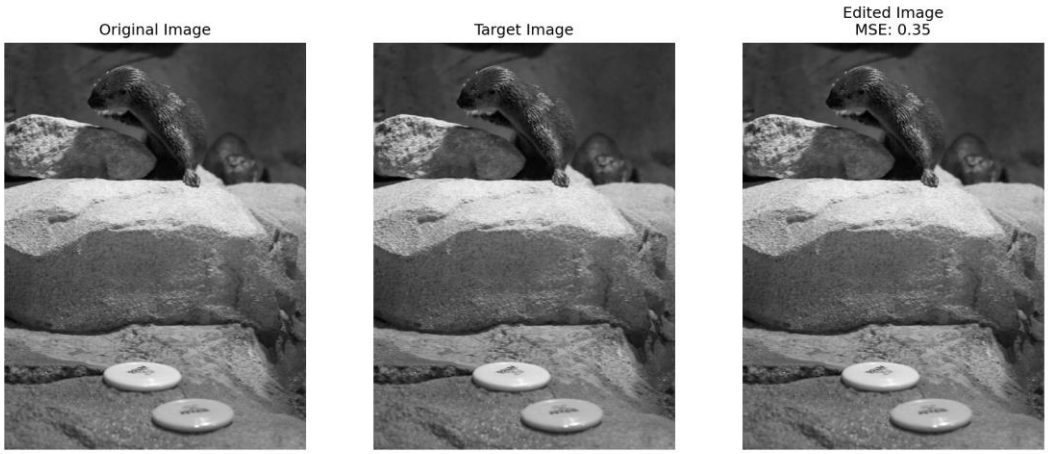


Image 9 - Sharpening Filter

Kernel used:

$$\begin{bmatrix} -0.25 & -0.25 & -0.25 \\ -0.25 & 3 & -0.25 \\ -0.25 & -0.25 & -0.25 \end{bmatrix}$$

Problem 2 – Biliteral Analysis

a) This function applies bilateral filtering to the given image

```
def clean_Gaussian_noise_bilateral(im, radius, stdSpatial, stdIntensity):
    # pre processing the image to calculate the bilateral filter
    im = im.astype(np.float64)
    rows, cols = im.shape
    cleanIm = np.zeros_like(im)
    x, y = np.meshgrid(np.arange(-radius, radius + 1), np.arange(-radius, radius + 1), indexing='ij')
    gs = np.exp(-(x**2 + y**2) / (2 * stdSpatial**2))
```

- So first of all as the notes said we make sure that we are working with float 64. And that's what we are doing in the code.
- After that we extract the rows and cols from the shape of the image, which help us iterate over each pixel later.
- We initialize 'cleanIm' as to store the final filtered image.
- Next we did a pre computing the spatial gaussian weights 'gs' using the given formula. for x and y we used the 'np.meshgrid', which creates a grid of coordinates representing the relative positions of neighboring pixels.

```
for i in range(rows):
    for j in range(cols):
```

- We used a double loop to iterate through each pixel (i,j) in the image.

```
# calculate the window of the image
x_min = max(0, i - radius)
x_max = min(rows, i + radius + 1)
y_min = max(0, j - radius)
y_max = min(cols, j + radius + 1)
window = im[x_min:x_max, y_min:y_max]
```

- (inside the double loop) For each pixel we need to define his local square window around It. We used the radius to define it.
- The way we defined the boundaries:
- ' x_min ' to prevents the window from going above of the image so it's the maximum between 'max(0,i-radius)' and 'x_max' prevents the window from going below the image it's the minimum between (rows, i+radius +1) .
- 'y_min' prevents the window from going left the image it's the maximum between (0,j-radius) and 'y_max' prevents from going right the image it's the minimum (cols, j+radius+1).
- ** Note the present of the image , x for vertical(rows) and y for horizontal (columns).

```
# calculate the bilateral filter for the window
gs_local = gs[:x_max - x_min, :y_max - y_min]
gi = np.exp(-((window - im[i, j])**2) / (2 * stdIntensity**2))
g = gs_local * gi
cleanIm[i, j] = np.sum(g * window) / np.sum(g)
```

- In this step (inside the double loop), we calculate the new pixel value from the given formula, but we break it into steps : first adjust 'gs' to match the current window second calculating 'gi', third combining weights 'g' and last calculating the new pixel value by applying the filter.

```
cleanIm = np.clip(cleanIm, 0, 255).astype(np.uint8)
return cleanIm
```

At the end (outside the double loop) we clipped the values to the range (0,255) and convert it back to unit8.

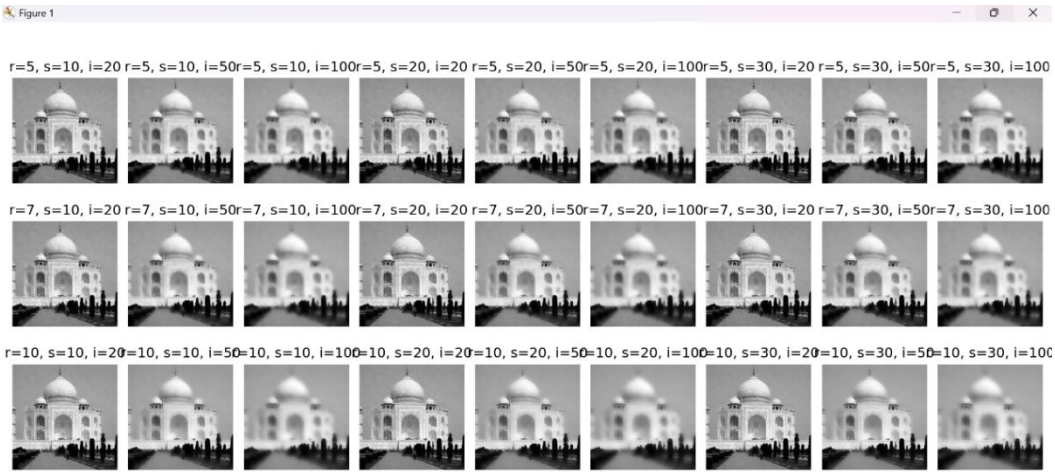
Section B

Taj image : our opinion bilateral filter can cleaned the image, we noticed that the image has a noise like salt and pepper noise. Applying a bilateral filter can help reduce the noise and make the image smoother while the details and edges preserved.

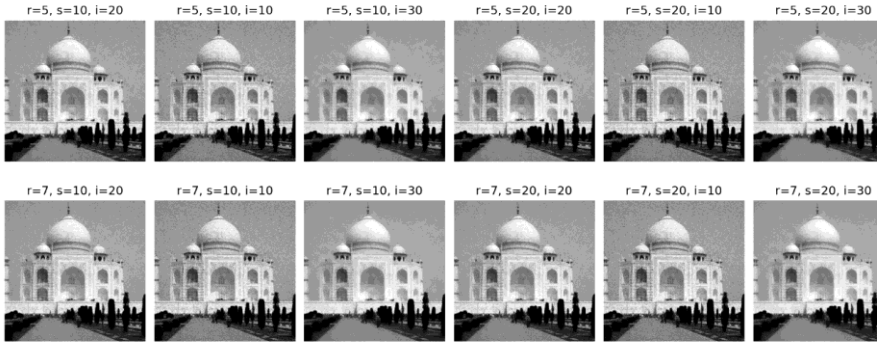
Choosing the right parameters:

We started by experimenting with different values, r=(5,7,10), stdSpatial=(10,20,30) and stdIntensity=(20,50,100).

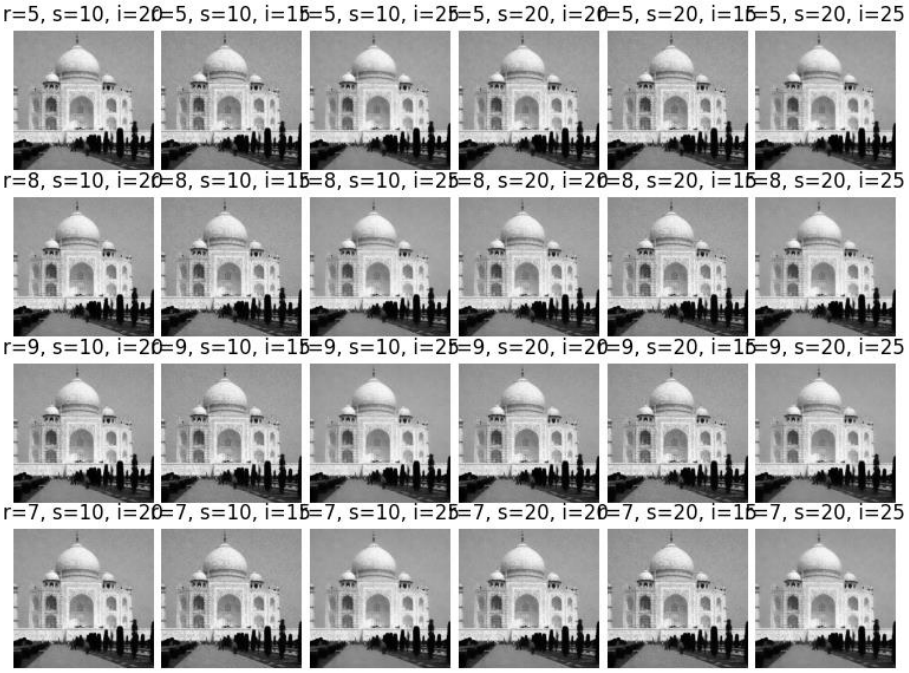
We noticed that stdIntensity =50/100 made the image overly blurred smoothing and stdIntensity=20 is the best. No noticeable difference was observed between stdSpatial=20 and 30. For radius no significant change was seen between r=7 and 10.



we then tried smaller stdIntensity values (10,20,30) and stdspatial =(10,20) , for the radius=(5,7)



From the outputs we can see that s=10, i=10 not good. So we try to do another values as we can see in the image :



We found the following parameters to be the best:

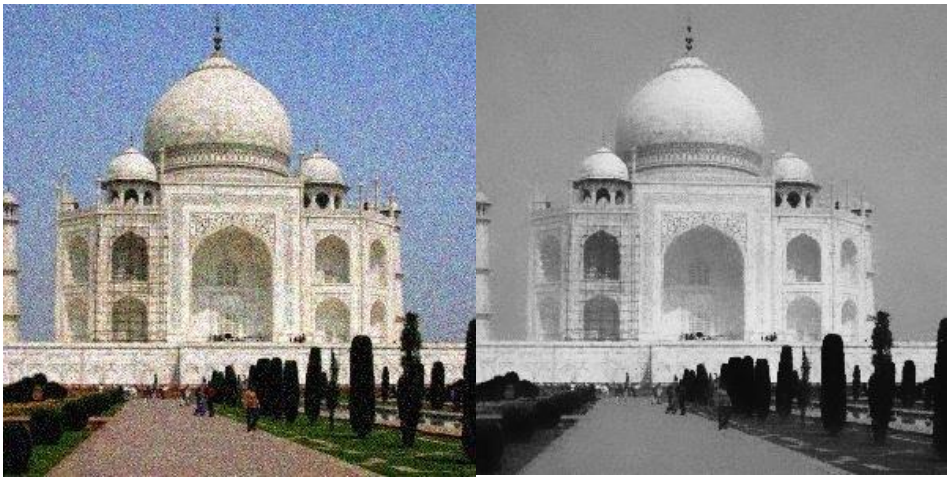
Radius=9

stdSpatial=20

stdIntensity=30

the result:

before **after**



The filter effectively enhanced the image and removed the noise while the edges and details where preserved.

We executed the same code adjusting the parameter values in the arrays as described above.

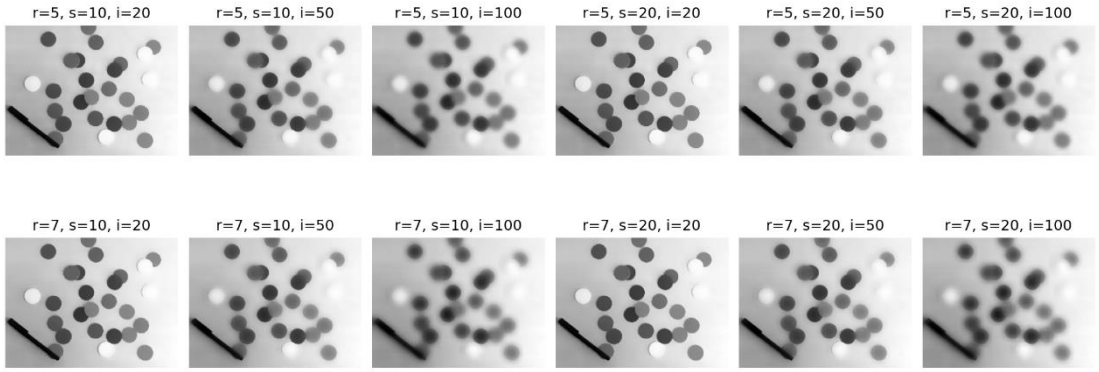
```
# Load the image
original_image_path = 'taj.jpg'
image = cv2.imread(original_image_path, cv2.IMREAD_GRAYSCALE)
# Define parameter ranges for testing
radius_values = [6,7,10,15]
stdSpatial_values = [20,40,60 ]
stdIntensity_values = [100]
# trying different parameter values
for radius in radius_values:
    for stdSpatial in stdSpatial_values:
        for stdIntensity in stdIntensity_values:
            processed_image = clean_Gaussian_noise_bilateral(image, radius, stdSpatial, stdIntensity)
            output_image_path = f"taj_r{radius}_s{stdSpatial}_i{stdIntensity}.jpg"
            cv2.imwrite(output_image_path, processed_image)
            result_images.append((processed_image, (radius, stdSpatial, stdIntensity)))
# plot the images and save them to compare the results
plt.figure(figsize=(15, 15))
for idx, (img, params) in enumerate(result_images):
    plt.subplot(len(radius_values), len(stdSpatial_values) * len(stdIntensity_values), idx + 1)
    plt.imshow(img, cmap='gray')
    plt.title(f"r={params[0]}, s={params[1]}, i={params[2]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Balls image:

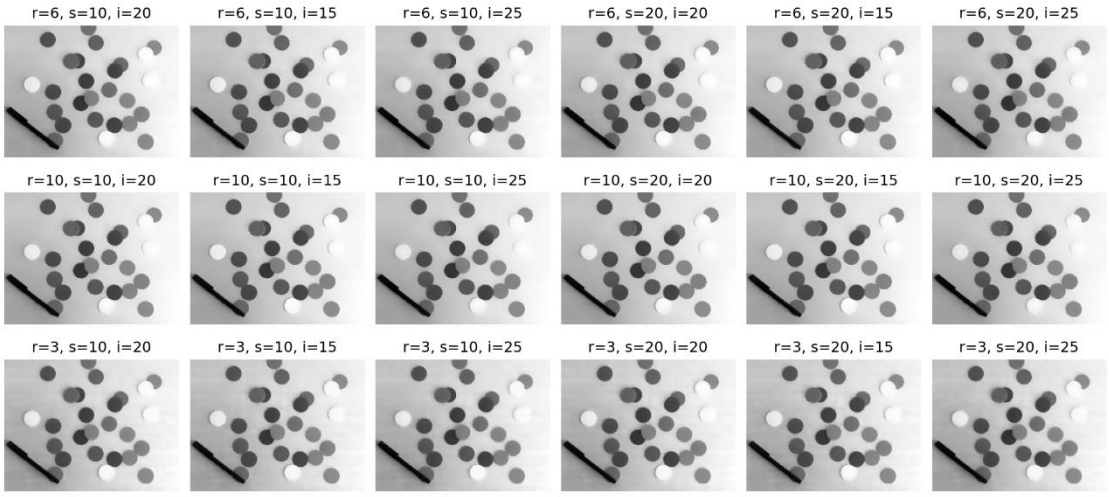
In our opinion the image already looking good however applying the bilateral filter can still help enhance the image smoothing a tiny noise while preserving the edges and details.

Choosing the right parameters:

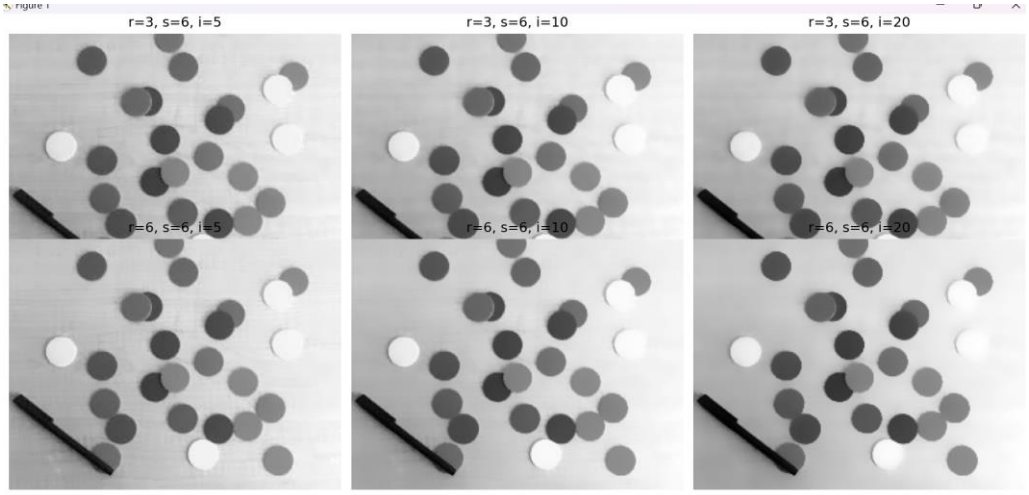
We started by experimenting with different values, r=(5,7), stdSpatial=(10,20) and stdIntensity=(20,50,100).



From the result, stdintensity=100 and 50 reduced the clarity so we tried another values 15,20,25 with radius=(3,6,10)



Last tried :we used the 2% of the diagnol to the stdspaital=$6.5 \approx 6$



We found the following parameters to be the best:

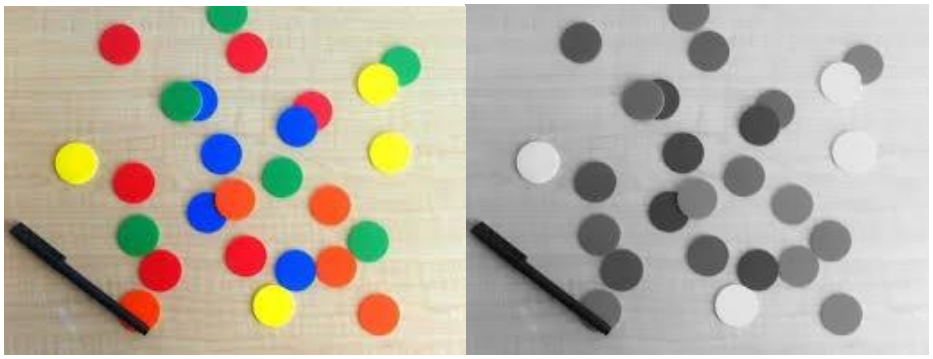
Radius=3

stdSpatial=6

stdIntensity=5

the result:

before **after**



After applying the bilateral filter the image looks smoother and the edges where preserved.

The code we executed (the same code as the taj image but we change the path and the values).

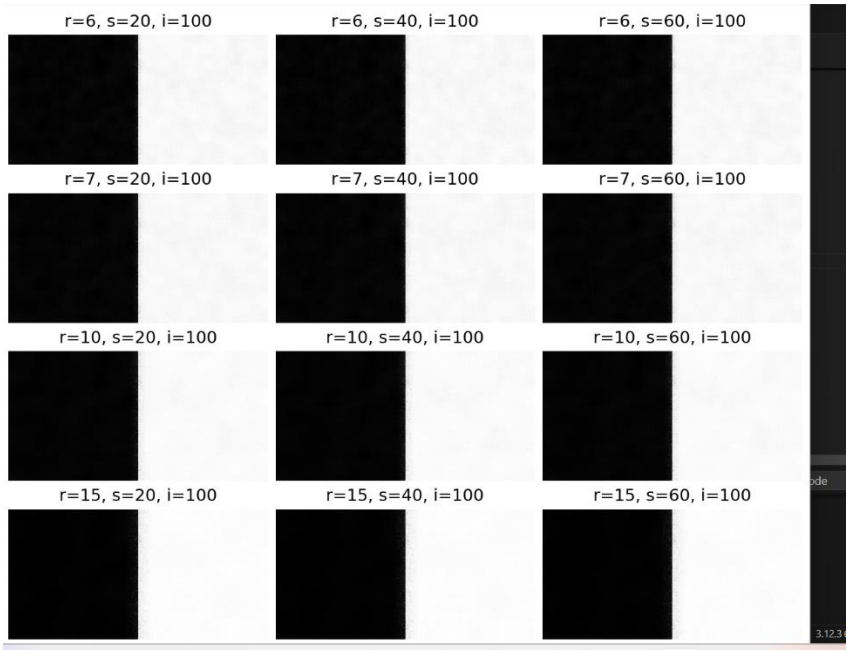
```
# Define parameter ranges for testing
radius_values = [3, 6]
stdSpatial_values = [6]
stdIntensity_values = [5, 10, 20]
# Initialize the result_images list
result_images = []
# Trying different parameter values
for radius in radius_values:
    for stdSpatial in stdSpatial_values:
        for stdIntensity in stdIntensity_values:
            processed_image = clean_Gaussian_noise_bilateral(image, radius, stdSpatial, stdIntensity)
            output_image_path = f"ball_r{radius}_s{stdSpatial}_i{stdIntensity}.jpg"
            cv2.imwrite(output_image_path, processed_image)
            result_images.append((processed_image, (radius, stdSpatial, stdIntensity)))

# Plot the images and save them to compare the results
plt.figure(figsize=(15, 15))
for idx, (img, params) in enumerate(result_images):
    plt.subplot(len(radius_values), len(stdSpatial_values) * len(stdIntensity_values), idx + 1)
    plt.imshow(img, cmap='gray')
    plt.title(f"r={params[0]}, s={params[1]}, i={params[2]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

The NoisyGray Image ,

the image contains strong salt pepper noise, and this noise affects the clarity of the original image due to that the bilateral filter well be effective in reducing the noise. Like the image before we start with initial values and we notice that stdIntensity=100 is the best between (20,50,100) after that we tried values (100,120,150) , the images became noisier with the 120 and 150 values so we kept 100 as optimal .

we tested a different stdspaital values (20,10,40,60) and we choose 60 as the best one for smoothing while preserving details.

For the radius we will choose 10. (also radius 15 is good but we didn't notice any different between the 2 values).



We found the following parameters to be the best:

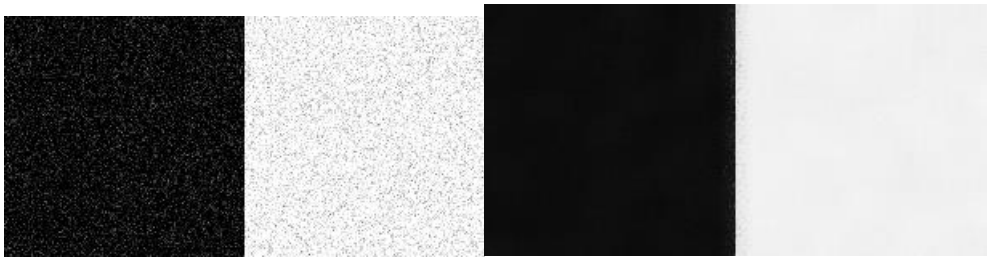
Radius=10

stdSpatial=60

stdIntensity=100

the result:

before **after**



From the result we can see that the filter significantly reduced the noise although some slight noise remains in the transition areas between black and white.

Problem 3 – Fix me

A) We began by applying the Biliteral filter as discussec in the previous question. the parameters used were:

stdSpatial: calculates 2% of the diagonal of the image, $0.02\sqrt{960^2 + 540^2} = 22.029 \approx 22$.

radius : various values were tested (5,10,15)

stdIntensity : various values were tested (20,50,100)

the following code was executed:

```
radius_values = [5,10,15]
stdIntensity_values = [20,50,100]
original_image_path = 'broken.jpg'
image = cv2.imread(original_image_path, cv2.IMREAD_GRAYSCALE)
for radius in radius_values:
    for stdIntensity in stdIntensity_values:
        clear_image_b = clean_Gaussian_noise_bilateral(image, radius, 22, stdIntensity)
        output_image_path = f"broken_edited_radius_{radius}_stdIntensity_{stdIntensity}.jpg"
        cv2.imwrite(output_image_path, clear_image_b)
        plt.figure()
        plt.imshow(clear_image_b, cmap="gray")
        plt.title(f"Radius: {radius}, Std Intensity: {stdIntensity}")
        plt.axis("off")
        plt.show()
```

Here are the results:

stdIntensity_100:

Radius	
5	

10	
15	

stdIntensity=50:

Radius	
5	

10	
15	

stdIntensity=20:

Radius	
5	

10	
15	

From the outputs we can conclude that using stdIntensity values of 100 and 50 does not produce good results as the image appears overly blurred. So stdIntensity=20 provides the best outcome among the three tested values. we tested additional 3 values (15,20,30) and we notice that 20 remains the optimal choice. For the radius value we didn't notice any significant differences among the 5,10 and 15 . we chose 10 for consistency.

We also observed that all the images exhibit salt and pepper noise , prompting us to test various filters from the class and tutorial on the original image (broken.jpg).

MinMax/ MaxMin filter:

Code executed:

```
original_image_path = 'broken.jpg'
image = cv2.imread(original_image_path, cv2.IMREAD_GRAYSCALE)
# min max
min_filtered_image = minimum_filter(image, size=3)
min_max_filtered_image = maximum_filter(min_filtered_image, size=3)
# max min
max_filtered_image = maximum_filter(image, size=3)
max_min_filtered_image = minimum_filter(max_filtered_image, size=3)
cv2.imwrite('min_max_filtered_image.jpg', min_max_filtered_image)
cv2.imwrite('max_min_filtered_image.jpg', max_min_filtered_image)
```

The result:

MaxMin :

MinMax:

The average/mean filter:

```
original_image_path = 'broken.jpg'
image = cv2.imread(original_image_path, cv2.IMREAD_GRAYSCALE)
average_image = cv2.blur(image, (3,3))
output_image_path = 'broken_edited_average.jpg'
cv2.imwrite(output_image_path, average_image)
```

The result :

The median filter:

```
original_image_path = 'broken.jpg'
image = cv2.imread(original_image_path, cv2.IMREAD_GRAYSCALE)
average_image = cv2.medianBlur(image, 3)
output_image_path = 'broken_edited_median.jpg'
cv2.imwrite(output_image_path, average_image)
```

The result:

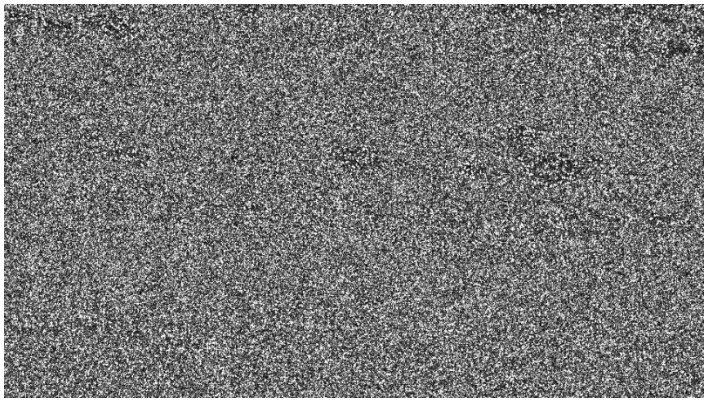
The blur gussian filter:

```
original_image_path = 'broken.jpg'
image = cv2.imread(original_image_path, cv2.IMREAD_GRAYSCALE)
average_image = cv2.GaussianBlur(image, (5,5),5)
output_image_path = 'broken_edited_gaussian.jpg'
cv2.imwrite(output_image_path, average_image)
```

The result:

Laplcian filter:

```
original_image_path = 'broken.jpg'
image = cv2.imread(original_image_path, cv2.IMREAD_GRAYSCALE)
laplacian = cv2.Laplacian(image, cv2.CV_64F)
laplacian_abs = cv2.convertScaleAbs(laplacian)
cv2.imwrite('laplacian_filtered_image.jpg', laplacian_abs)
```



Based on the outputs/results the median filter was most effective for addressing salt and pepper noise (and thats support what we learned lecture 5).

We tested between kernel size=3/5/7 and we noticed that 5 is the best.

To improve the image we tested combinations of the bilateral filter with median filter:

Bilateral filter followed by median filter:

```
# Apply Bilateral Filter
image = cv2.imread("broken.jpg", cv2.IMREAD_GRAYSCALE)
clear_image_b = clean_Gaussian_noise_bilateral(image, 10, 22, 20)
output_image_path = "a_after_bilateral.jpg"
cv2.imwrite(output_image_path, clear_image_b)
# Apply Median Filter
image = cv2.imread("a_after_bilateral.jpg", cv2.IMREAD_GRAYSCALE)
clear_image_b = cv2.medianBlur(image, 5)
output_image_path = "a.jpg"
cv2.imwrite(output_image_path, clear_image_b)
```

Median filter followed by bilateral filter:

```
# Apply Median Filter
image = cv2.imread("broken.jpg", cv2.IMREAD_GRAYSCALE)
clear_image_b = cv2.medianBlur(image, 5)
output_image_path = "a_after_median.jpg"
cv2.imwrite(output_image_path, clear_image_b)
# Apply Bilateral Filter
image = cv2.imread("a_after_median.jpg", cv2.IMREAD_GRAYSCALE)
clear_image_b = clean_Gaussian_noise_bilateral(image, 10, 22, 20)
clear_image_b = cv2.medianBlur(image, 5)
output_image_path = "a_medianfirst.jpg"
cv2.imwrite(output_image_path, clear_image_b)
```

We observed that applying the bilateral filter first followed by the median filter provided better result.

Next , we experimented with applying smoothing/ sharpeness: (on the image result after applying bilateral followed by median)

```
sharpen_kernel = np.array([[0, -1, 0],
                           [-1, 5, -1],
                           [0, -1, 0]])
smoothing_kernel = np.ones((3, 3), np.float32) / 9
image = cv2.imread("broken.jpg", cv2.IMREAD_GRAYSCALE)
image_gussianed = clean_Gaussian_noise_bilateral(image, 10, 22, 20)
image_medianed = cv2.medianBlur(image_gussianed, 5)
image_sharpened = cv2.filter2D(image_medianed, -1, sharpen_kernel)
image_smoothed = cv2.filter2D(image_medianed, -1, smoothing_kernel)
cv2.imwrite("a_sharpened.jpg", image_sharpened)
cv2.imwrite("a_smoothed.jpg", image_smoothed)
```

Smothing:

Sharppening:

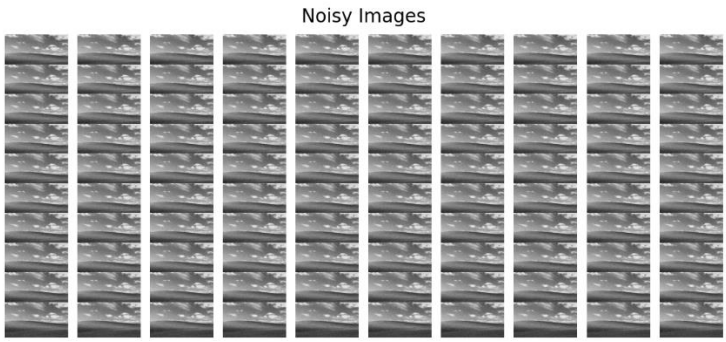
the sharpening approach did not yield satisfactory outputs and the smoothing techniques showed only marginal improvements. To conclusion the best method involves applying the bilateral filter first to reduce gussian noise followed by the median filter to clean salt and pepper noise.

B) using 'noised_images.npy' file. We began to understand what the file included , we loaded the file using numpy.load. revealing an array of shape (100, height , width), after checking it containg 100 noisy images.

```
35  # part B
36  # Load the noisy images
37  noised_images = np.load("noised_images.npy")
38  # Number of images
39  num_images = noised_images.shape[0]
40  print('the number of images:', num_images)
41
```

TERMINAL PORTS SPELL CHECKER 9 COMMENTS OUTPUT DEBUG CONSOLE PROBLEMS

```
100

[Done] exited with code=0 in 1.532 seconds

[Running] python -u "c:\Users\נ'אנ\Desktop\ImageProcessing2024_2025_HW3\q3\q3.py"
the number of images: 100

[Done] exited with code=0 in 1.436 seconds

[Running] python -u "c:\Users\נ'אנ\Desktop\ImageProcessing2024_2025_HW3\q3\q3.py"
the number of images: 100

[Done] exited with code=0 in 1.412 seconds
```



To reduce the noise , the median value was computes for each pixel across all the 100 images.

```
# part B
# Load the noisy images
noised_images = np.load("noised_images.npy")
stacked_median = np.median(noised_images, axis=0).astype(np.uint8)
cv2.imwrite("b.jpg", stacked_median)
```

The best result with median: