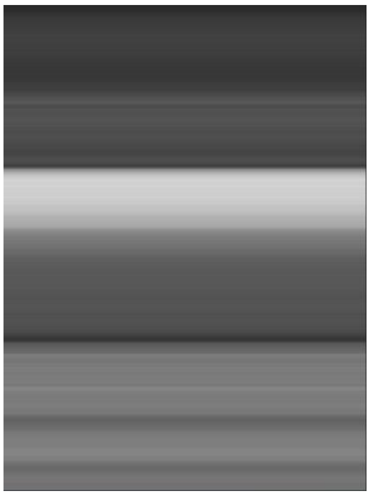
Ilan Sapoznikov

Liad Arvatz

Image Processing: Assignment #3

Problem 1- What happened here:

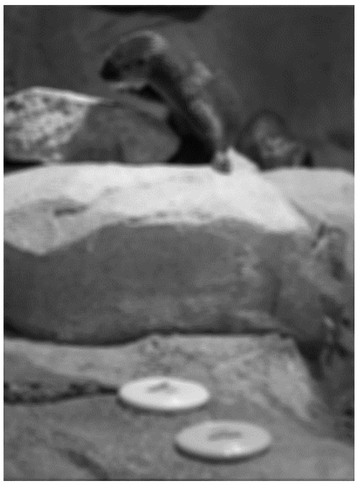
Image 1:



mean filter with a vertical kernel

A mean filter with a vertical kernel can be particularly useful for addressing images that exhibit a horizontal spread from the original. By applying a rectangular mask and employing mean convolution, we aim to achieve the desired result.

Image 2:

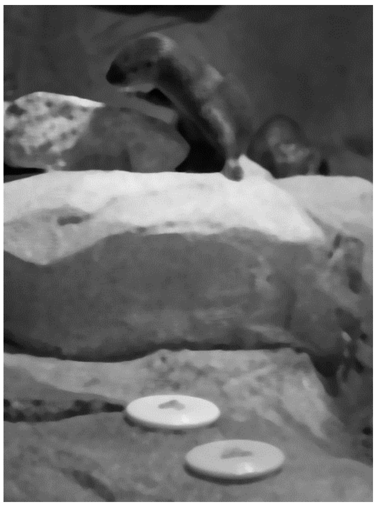


Mean filter

In using the mean filter, we observe a blurring effect without loss of information, unlike the median mask. When employing the median mask, objects appear smoother because the pixel with the highest frequency is selected.

In contrast, the mean filter calculates the average of neighboring pixels, distributing equal weight to each. As a result, significant smoothing is not noticeable in the image.

Image 3:

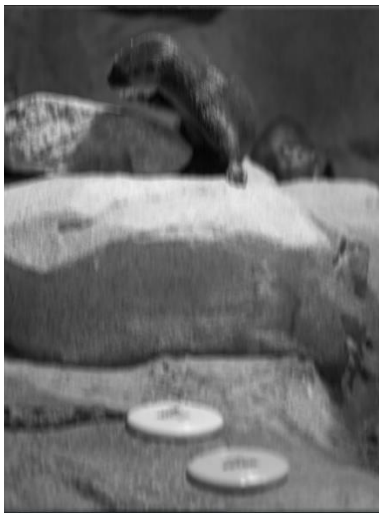


Median filter

Blurring by taking a local environment and finding the median within it causes visible issues in high-frequency areas. When we compute the median within a specific region, we effectively select a point that represents the majority. However, in areas characterized by rapid changes between pixels, this method tends to remove such changes.

This phenomenon is evident in the picture of the otter's body, where regions exhibit both light and dark areas. Here, the high frequency of changes is diminished, resulting in a loss of detail.

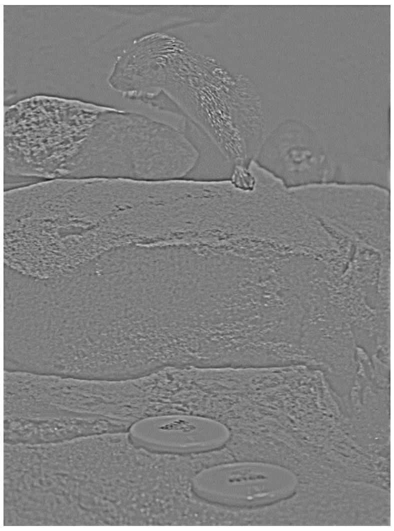
Image 4:



Convolution with a vertical mask

Convolution with a vertical mask results in vertical smearing, as evidenced by the image.

Image 5:



Convolution with gaussian mask

When convolving with a Gaussian mask, it's important to note that the significant information in this image resides within the edges.

We can see that we took G- a blurring mask and them make our picture more blur with convolution operation B*G. Then we decrease the sharp image B from B blur. and then we got the image with the sharp part as we can see in image.

For example:

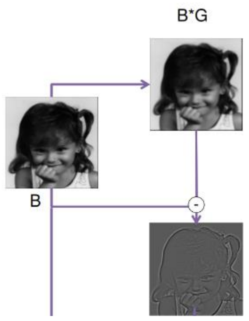
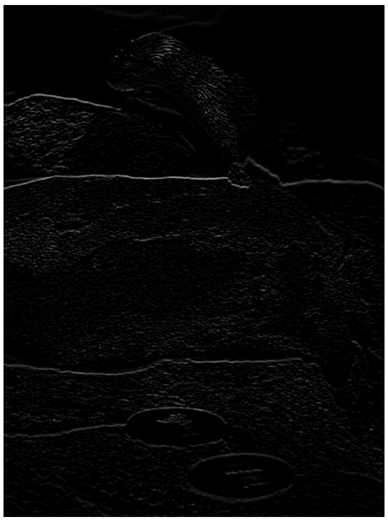


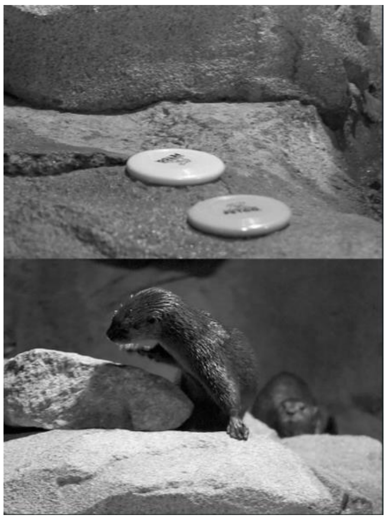
Image 6:



Laplacian filter

The Laplacian filter is designed to detect abrupt intensity transitions within an image, effectively highlighting its edges. In the provided image, we observe the distinct edges of the objects, exemplifying the functionality of the Laplacian Filter.

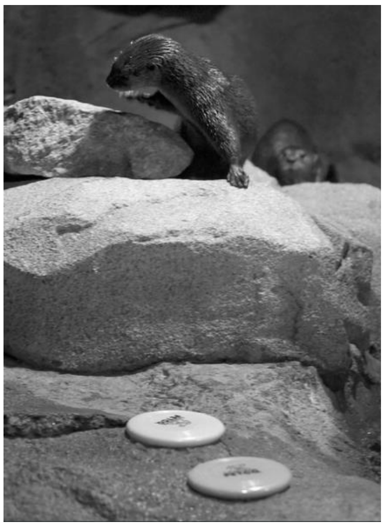
Image 7:



Cyclic convolution/Cyclic padding

Cyclic convolution, also known as cyclic padding, employs a delta mask wherein the white pixel is not positioned at the center but rather in a lower part of the mask. Consequently, the information, or "energy," of the lower lines is propagated upwards, as depicted in this picture.

Image 8:



Convolution with delta mask

We observe that the image has not undergone any change, suggesting that the operation performed is a convolution with a delta mask. This mask is composed of black pixels except for one white pixel in the center, ensuring that the image remains unchanged.

For example:

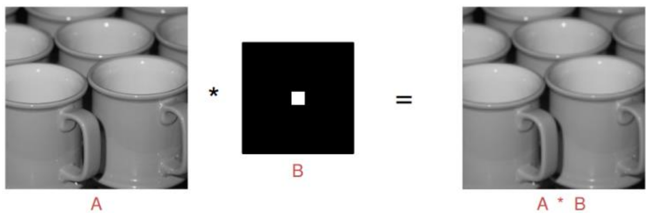
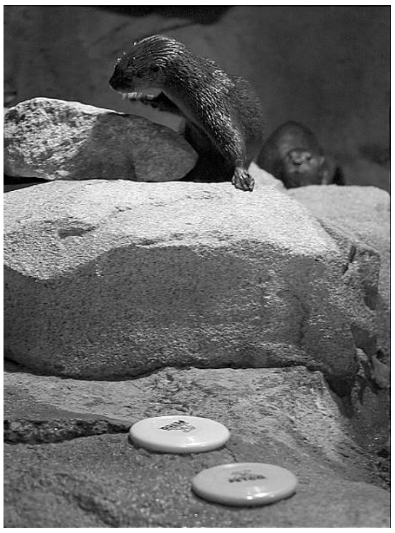


Image 9:



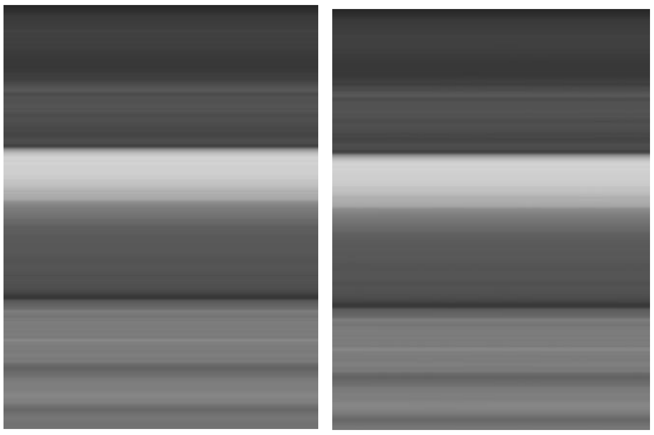
<u>Sharpening</u>

If we follow the steps described for convolution with a Gaussian mask, we obtain an image with pronounced edges. By adding this to the original image, the edges of the original image become sharper, as the process emphasizes them. This is precisely what is observed in this picture.

Bonus:

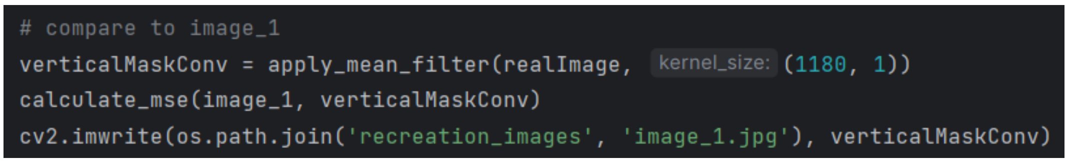
The original images is shown on the left, and our images is shown on the right. We will note that most of the parameters were determined through trial and error on our part.

Image 1:



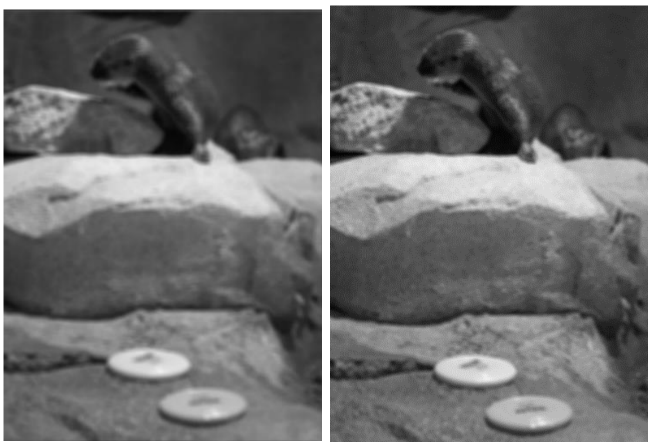
We used mean filter with horizontal convolution mask size (1180,1)

MSE- 0.11



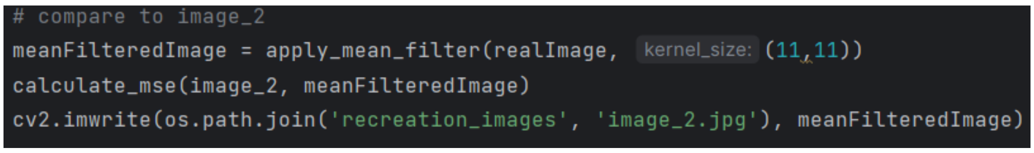
```
# compare to image_1
verticalMaskConv = apply_mean_filter(realImage, kernel_size: (1180, 1))
calculate_mse(image_1, verticalMaskConv)
cv2.imwrite(os.path.join('recreation_images', 'image_1.jpg'), verticalMaskConv)
```

Image 2:



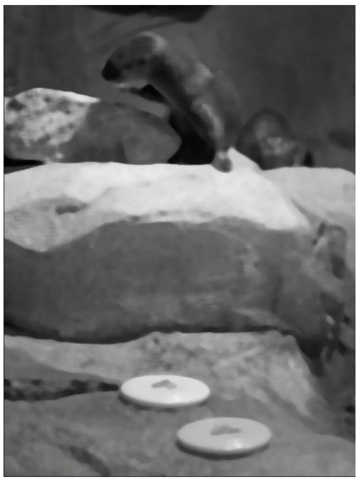
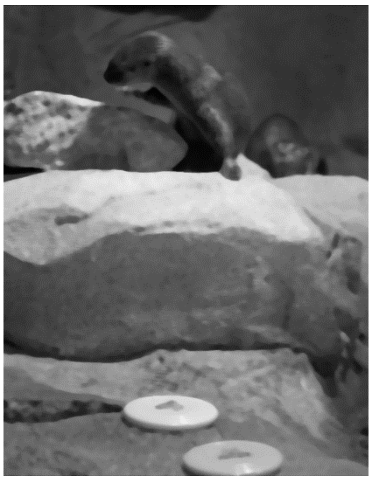
We used mean filter with square convolution mask size (11,11)

MSE- 12.87



```
# compare to image_2
meanFilteredImage = apply_mean_filter(realImage, kernel_size: (11,11))
calculate_mse(image_2, meanFilteredImage)
cv2.imwrite(os.path.join('recreation_images', 'image_2.jpg'), meanFilteredImage)
```

Image 3:

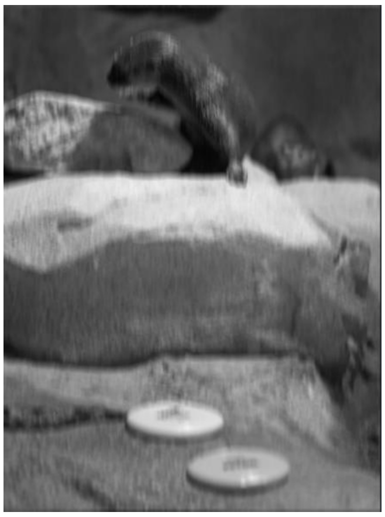
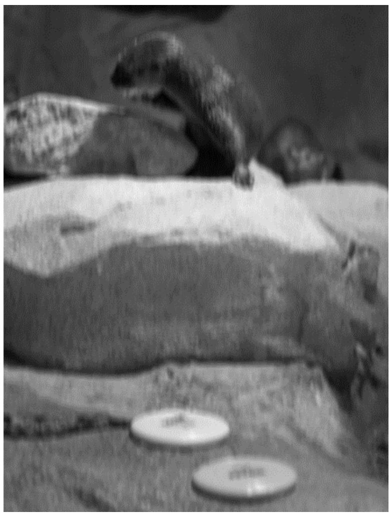



Looks smoother and blurred. It also can be average filter, gaussian filter, median filter and bilateral filter.

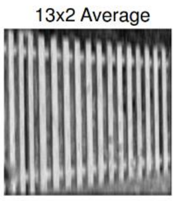
We used median filter with square convolution mask size (9,9)

MSE- 4.24

Image 4:

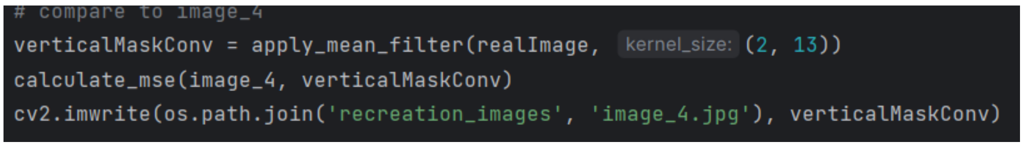



We noticed the image has the same blur as we saw in lecture:



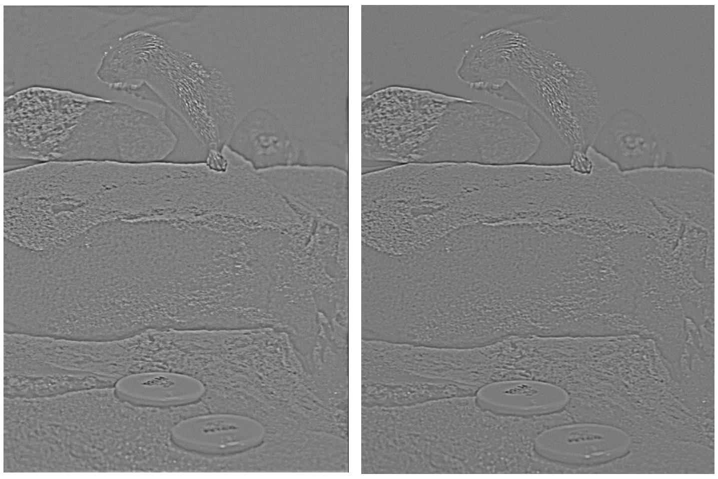
We used mean filter with square convolution mask size (13,2)

MSE- 7.44



```
# compare to image_4
verticalMaskConv = apply_mean_filter(realImage, kernel_size: (2, 13))
calculate_mse(image_4, verticalMaskConv)
cv2.imwrite(os.path.join('recreation_images', 'image_4.jpg'), verticalMaskConv)
```

Image 5:



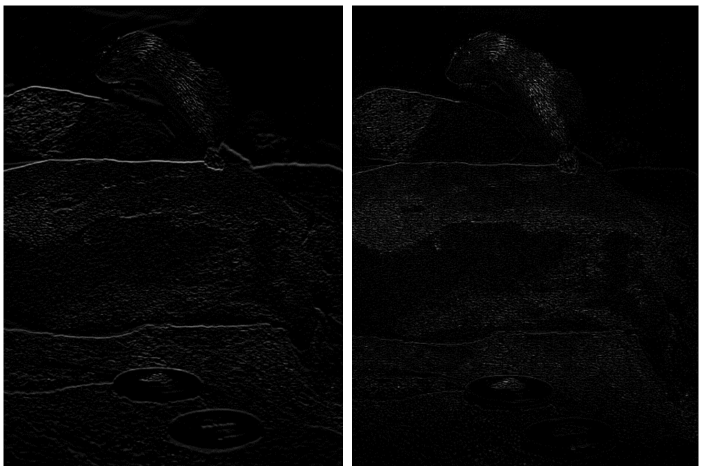
We used gaussian mask with size (9,9) and sigma 10.

Suppose B is the pixel's matrix of the original matrix.

As we learned, to get this image, we did B-B*kernel

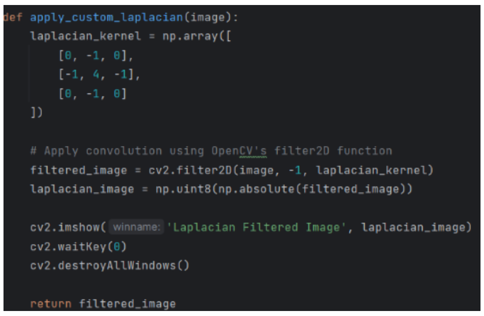
MSE- 20.77

Image 6:



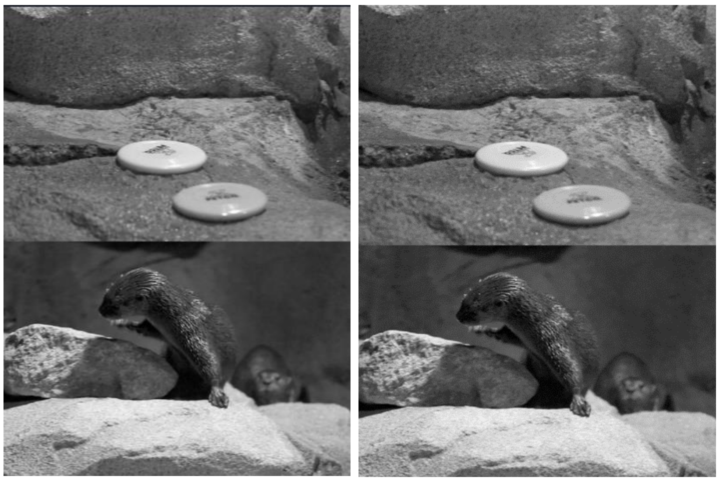
We used derivation kernel- [[0, -1, 0], [-1, 4, -1], [0, -1, 0]], as we learned to get laplacian filter.

MSE- 39.72



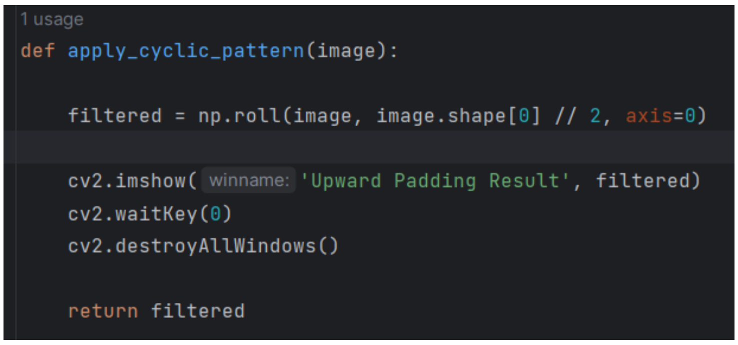
```python
def apply_custom_laplacian(image):
    laplacian_kernel = np.array([
        [0, -1, 0],
        [-1, 4, -1],
        [0, -1, 0]
    ])

    # Apply convolution using OpenCV's filter2D function
    filtered_image = cv2.filter2D(image, -1, laplacian_kernel)
    laplacian_image = np.uint8(np.absolute(filtered_image))

    cv2.imshow( winname: 'Laplacian Filtered Image', laplacian_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    return filtered_image
```

Image 7:



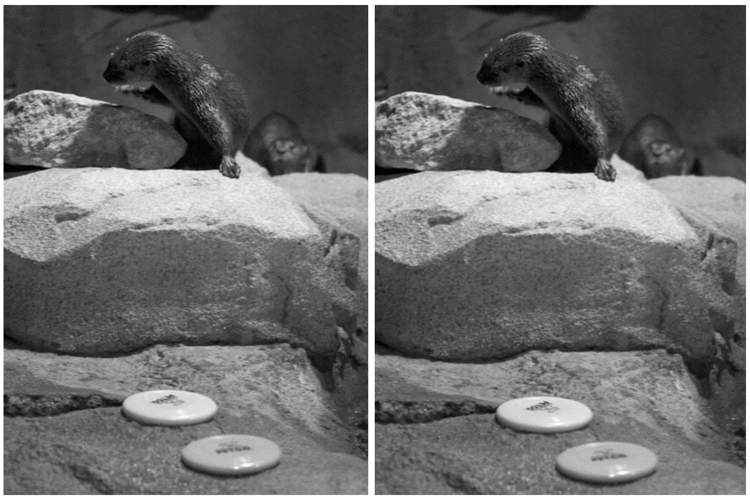
We used np.roll to do cyclic roll by cut and shaped the button half of the image.

MSE-1.18



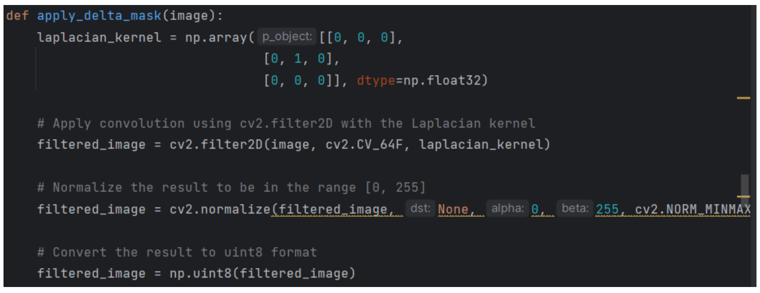
```
1 usage
def apply_cyclic_pattern(image):

    filtered = np.roll(image, image.shape[0] // 2, axis=0)

    cv2.imshow( winname: 'Upward Padding Result', filtered)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    return filtered
```

Image 8:



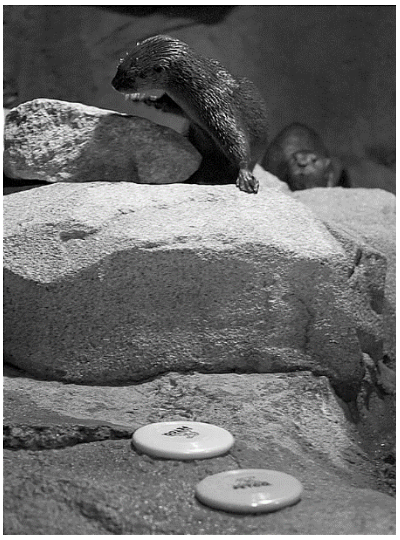
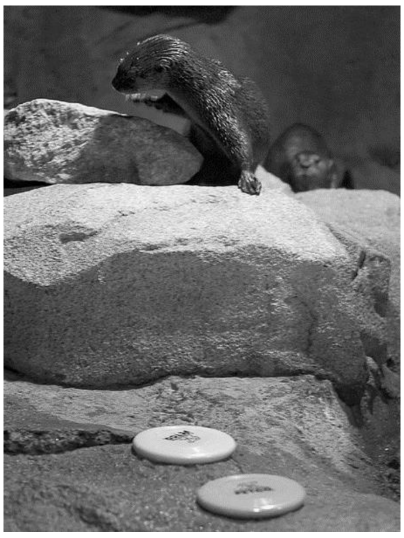
We used delta mask to do a convolution. The convolution doesn't make any sense to the image as we expected.

MSE-0.34



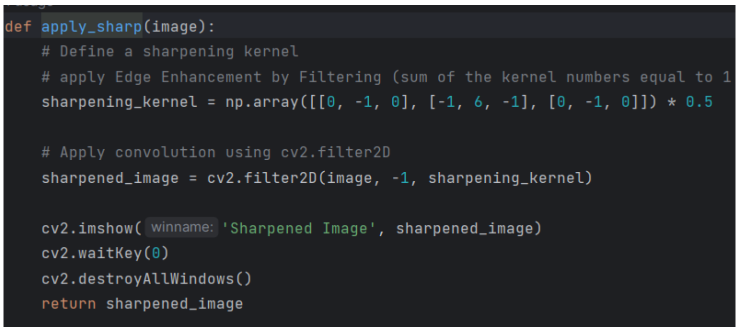
```
def apply_delta_mask(image):
    laplacian_kernel = np.array( p_object: [[0, 0, 0],
                                 [0, 1, 0],
                                 [0, 0, 0]], dtype=np.float32)

    # Apply convolution using cv2.filter2D with the Laplacian kernel
    filtered_image = cv2.filter2D(image, cv2.CV_64F, laplacian_kernel)

    # Normalize the result to be in the range [0, 255]
    filtered_image = cv2.normalize(filtered_image,  dst: None,  alpha: 0,  beta: 255, cv2.NORM_MINMAX

    # Convert the result to uint8 format
    filtered_image = np.uint8(filtered_image)
```

Image 9:

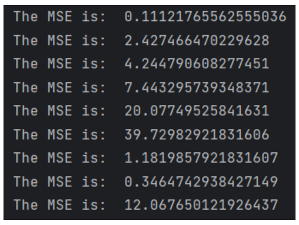
We used gaussian mask to sharp the image. we used it because we notch the image is more sharper than the original one, so we want to focus more on the edges. This is what gaussian mask doe's.

MSE- 12.06



```
def apply_sharp(image):
    # Define a sharpening kernel
    # apply Edge Enhancement by Filtering (sum of the kernel numbers equal to 1
    sharpening_kernel = np.array([[0, -1, 0], [-1, 6, -1], [0, -1, 0]]) * 0.5

    # Apply convolution using cv2.filter2D
    sharpened_image = cv2.filter2D(image, -1, sharpening_kernel)

    cv2.imshow( winname: 'Sharpened Image', sharpened_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    return sharpened_image
```

The MSEs:



```
The MSE is:  0.11121765562555036
The MSE is:  2.427466470229628
The MSE is:  4.244790608277451
The MSE is:  7.443295739348371
The MSE is:  20.07749525841631
The MSE is:  39.72982921831606
The MSE is:  1.1819857921831607
The MSE is:  0.3464742938427149
The MSE is:  12.067650121926437
```

Problem 2- Biliteral analysis:

In the BilateralCleaning.py there are 2 functions implemented. The first function is a step-by-step implementation of the calculations described in the assignment, and the second one is a refined version of it. The second function has a slight modification in the calculation of gs, which yielded better results. Therefore, we used it for corrections in clean_gaussian_noise_bilateral, while still keeping both functions nonetheless.

Below we will describe the first version of the function and we will show the major changes in the second function.

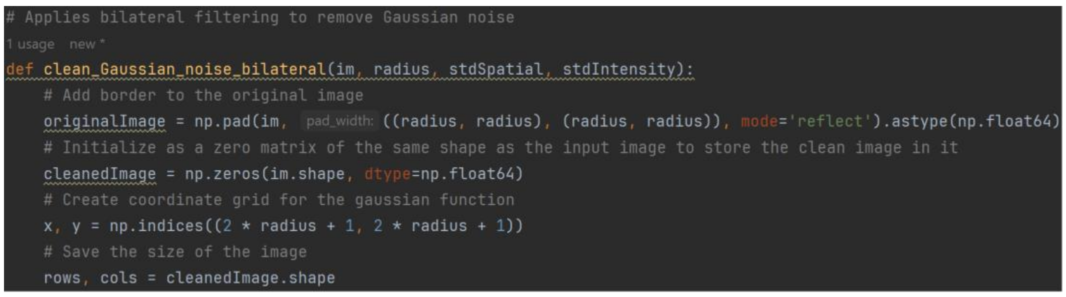
Code explanation:

This code performs image denoising using bilateral filtering, which is a technique commonly used for noise reduction while preserving edges in images.

First of all, this is an overview of the parameters that the clean_Gaussian_noise_bilateral function receives:

- **im**: The input grayscale image as a 2D NumPy array.
- **radius**: The radius of the square kernel used for filtering. This determines the size of the neighborhood over which the filtering is performed. A larger radius will consider more pixels in the computation, leading to stronger smoothing but at the cost of higher computational complexity.
- **stdSpatial**: The standard deviation of the spatial component of the filter. This parameter controls the extent to which the distance between pixels influences their weight in the averaging process. A larger value will make the spatial weight decrease more slowly, leading to more smoothing.
- **stdIntensity**: The standard deviation of the intensity component of the filter. This parameter controls how much the difference in intensity between the central pixel and its neighbors affects the weighting. A larger value will make the filter more forgiving towards differences in intensity, leading to stronger smoothing of features with high intensity variance.

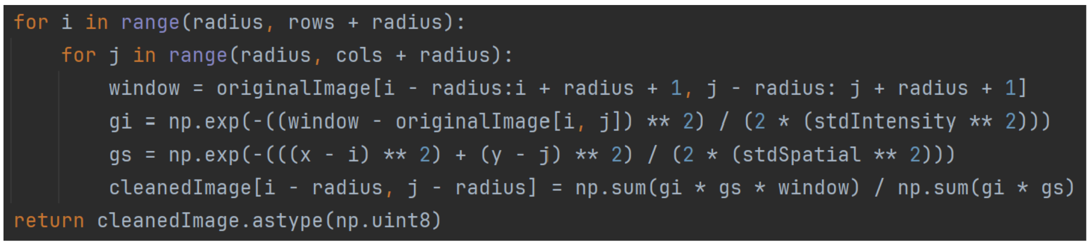
This is a break down of the code step by step (<u>first version</u>):



```
# Applies bilateral filtering to remove Gaussian noise
1 usage  new *
def clean_Gaussian_noise_bilateral(im, radius, stdSpatial, stdIntensity):
    # Add border to the original image
    originalImage = np.pad(im,  pad_width: ((radius, radius), (radius, radius)), mode='reflect').astype(np.float64)
    # Initialize as a zero matrix of the same shape as the input image to store the clean image in it
    cleanedImage = np.zeros(im.shape, dtype=np.float64)
    # Create coordinate grid for the gaussian function
    x, y = np.indices((2 * radius + 1, 2 * radius + 1))
    # Save the size of the image
    rows, cols = cleanedImage.shape
```

The function first uses the ‘pad’ function to pad the input image with reflected borders to handle edge cases.
Then It initializes an empty matrix to store the filtered image, which is the output of this function.

We use the ‘indices’ function to create a grid of (x,y) coordinates. In that way, for each pixel in the image, we have the corresponding (x, y) coordinates that represent its spatial location in the image. These coordinates are used to compute the spatial component of the Gaussian function in bilateral filtering, which helps determine the contribution of neighboring pixels to the filtering process.

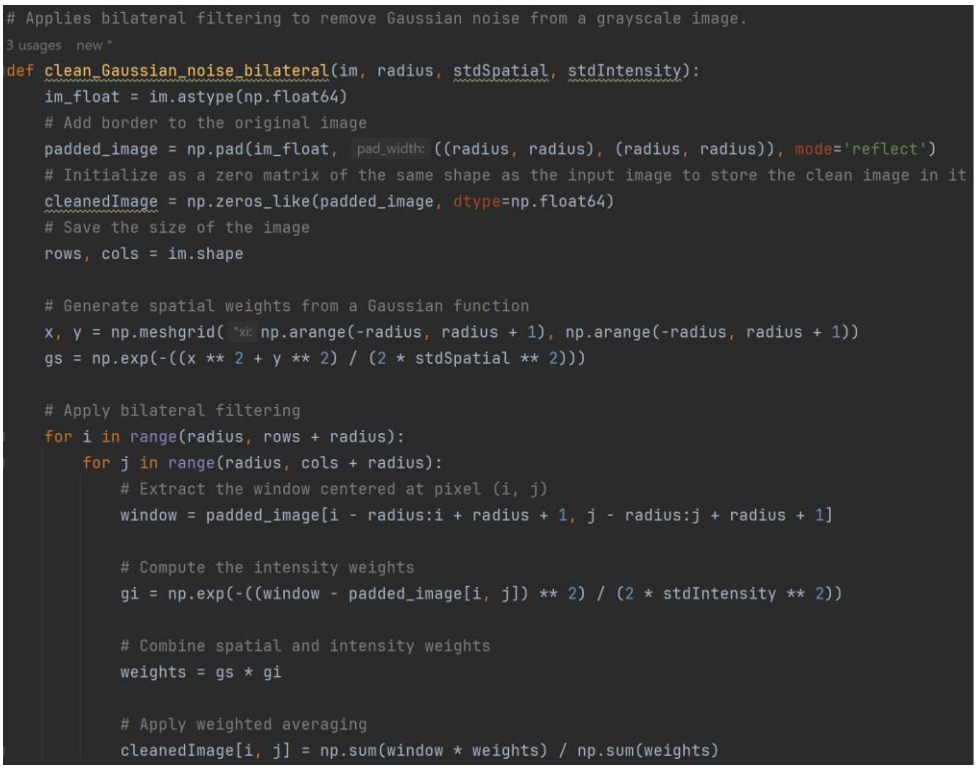


```python
for i in range(radius, rows + radius):
    for j in range(radius, cols + radius):
        window = originalImage[i - radius:i + radius + 1, j - radius: j + radius + 1]
        gi = np.exp(-((window - originalImage[i, j]) ** 2) / (2 * (stdIntensity ** 2)))
        gs = np.exp(-(((x - i) ** 2) + (y - j) ** 2) / (2 * (stdSpatial ** 2)))
        cleanedImage[i - radius, j - radius] = np.sum(gi * gs * window) / np.sum(gi * gs)
return cleanedImage.astype(np.uint8)
```

Afterward, we traverse through every pixel of the original image within the bordered image. The algorithm computes the value of each new pixel according to the instructions provided in the document. This involves systematically visiting each pixel location and applying the specified calculations to determine the updated pixel value based on the surrounding pixels. The process iterates until all pixels in the bordered image have been processed, and their new values have been computed according to the prescribed algorithmic steps.

The major change we did in the <u>second version</u> of the function is the way we calculated gs. In the second version the spatial weights (gs) are precomputed before the loop based on the distance from the center pixel within the kernel window. This approach assumes a fixed kernel center for spatial weight calculation, which is a common practice in bilateral filtering. This change led to an improvement in the output we received.

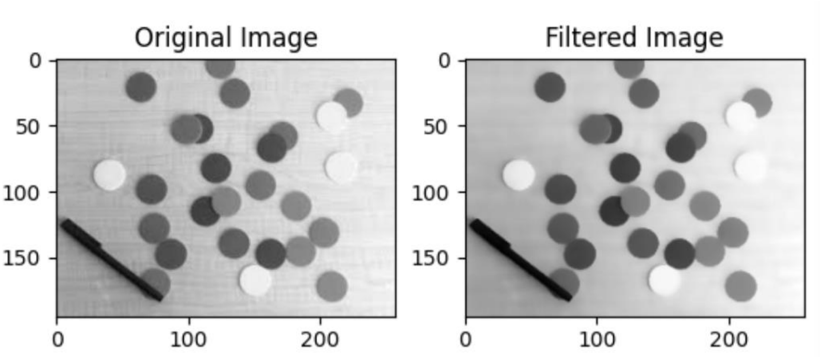
Below is our code of the second (and final) version:



```python
# Applies bilateral filtering to remove Gaussian noise from a grayscale image.
3 usages  new *
def clean_Gaussian_noise_bilateral(im, radius, stdSpatial, stdIntensity):
    im_float = im.astype(np.float64)
    # Add border to the original image
    padded_image = np.pad(im_float,  pad_width: ((radius, radius), (radius, radius)), mode='reflect')
    # Initialize as a zero matrix of the same shape as the input image to store the clean image in it
    cleanedImage = np.zeros_like(padded_image, dtype=np.float64)
    # Save the size of the image
    rows, cols = im.shape

    # Generate spatial weights from a Gaussian function
    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)))

    # Apply bilateral filtering
    for i in range(radius, rows + radius):
        for j in range(radius, cols + radius):
            # Extract the window centered at pixel (i, j)
            window = padded_image[i - radius:i + radius + 1, j - radius:j + radius + 1]

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

            # Combine spatial and intensity weights
            weights = gs * gi

            # Apply weighted averaging
            cleanedImage[i, j] = np.sum(window * weights) / np.sum(weights)
```

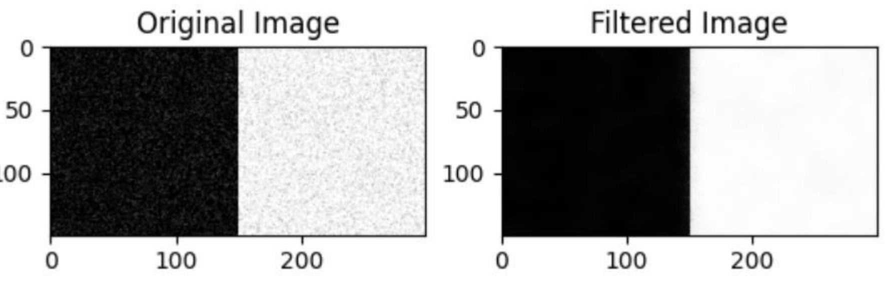
Final Results:

1. ‘balls.jpg’ with the parameters [3,40,20]:



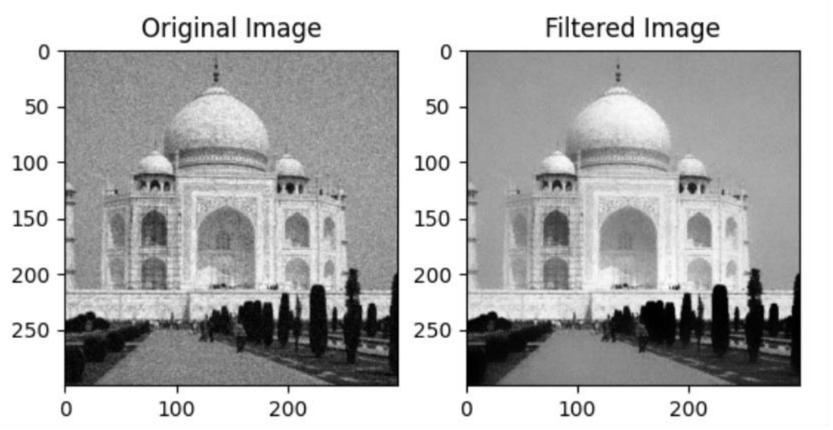
This image contains distinct artifacts attributed to JPEG compression, which affect details differently compared to typical salt and pepper or static noises found in the other images. Despite these differences, employing parameters [3, 40, 20] effectively smoothed out the artifacts while still preserving the intricate details within the image. This suggests that the algorithm is robust enough to handle various types of noise and artifacts, providing satisfactory results even in challenging scenarios like JPEG compression artifacts.

2. ‘NoisyGrayImages.png’ with the parameters [15,6.7,100]:



The filter demonstrates exceptional performance in effectively reducing salt and pepper noise while retaining sharp edges and color fidelity. Through experimentation, it was determined that using the values [15,6.7,100] as the parameters yielded the most optimal preservation of the image's details. “stdSpatial” value calculated by calculating 2 percent of image diagonal ($stdSaptial = 0.02 * image\_diagonal$).
This combination effectively balances noise reduction with the preservation of important image features, resulting in superior image quality.

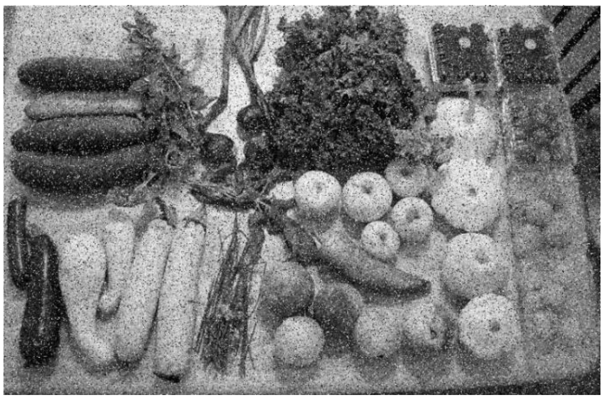
3. ‘taj.jpg’ with the parameters [15,8.48,30]:



After careful consideration, it was observed that using the parameters [15, 8.48, 30] notably reduced the visibility of noise while retaining finer details such as edges and intricate features like bushes and building walls. As in the previous image, the middle parameter (stdSpatial) was calculated in the same way.
This parameter configuration strikes a balance between noise reduction and detail preservation, resulting in a more satisfactory outcome for the image.

Problem 3- Fix Me!:

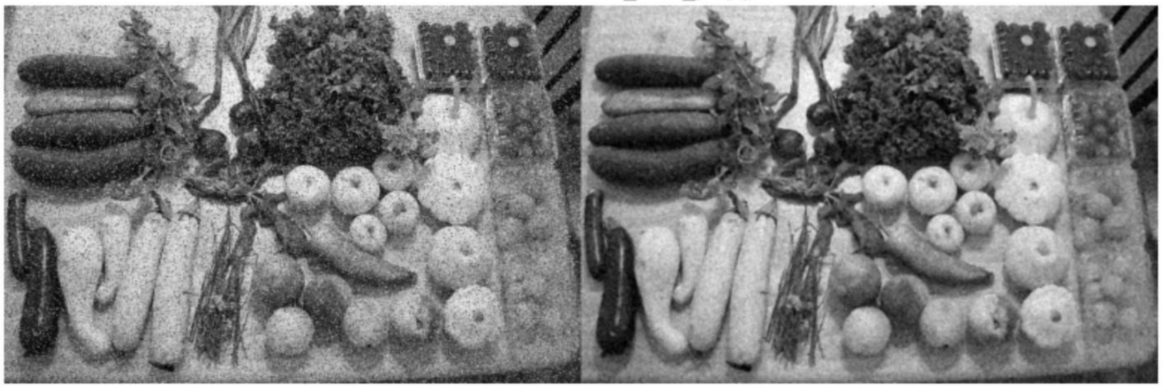
a. Below is the original “broken” image:



To enhance the clarity of the image, we performed the following steps:

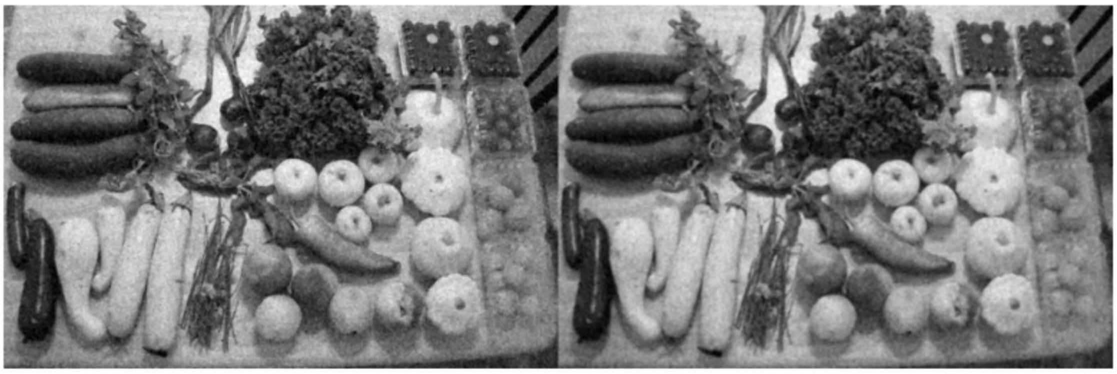
1. First and foremost, we addressed the salt-and-pepper noise by employing a median mask convolution algorithm. This algorithm effectively removes salt-and-pepper noise while preserving the integrity of the image's features.

with VS without salt_and_pepper



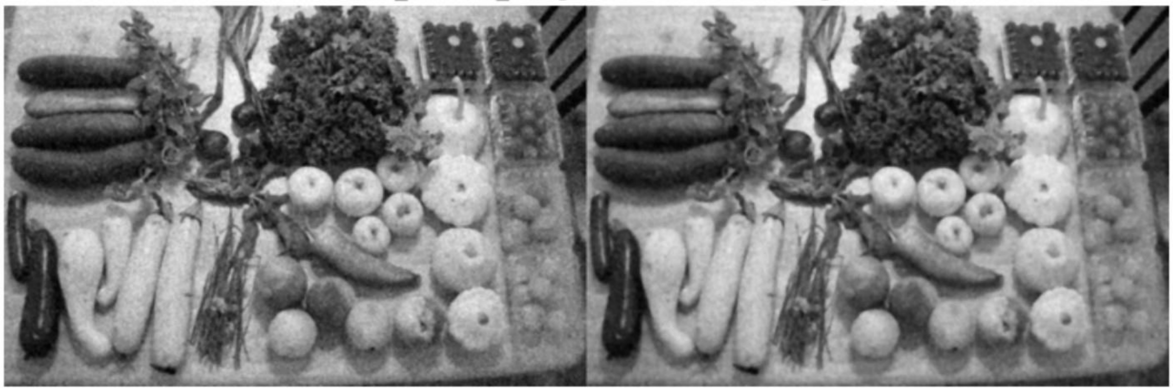
2. Next, we applied convolution using a mean mask. This technique calculates the average value of neighboring pixels, thereby smoothing out minor imperfections and enhancing overall image quality. We opted for a kernel size of 3 to avoid excessive blurring of the image.

before & after mean filter



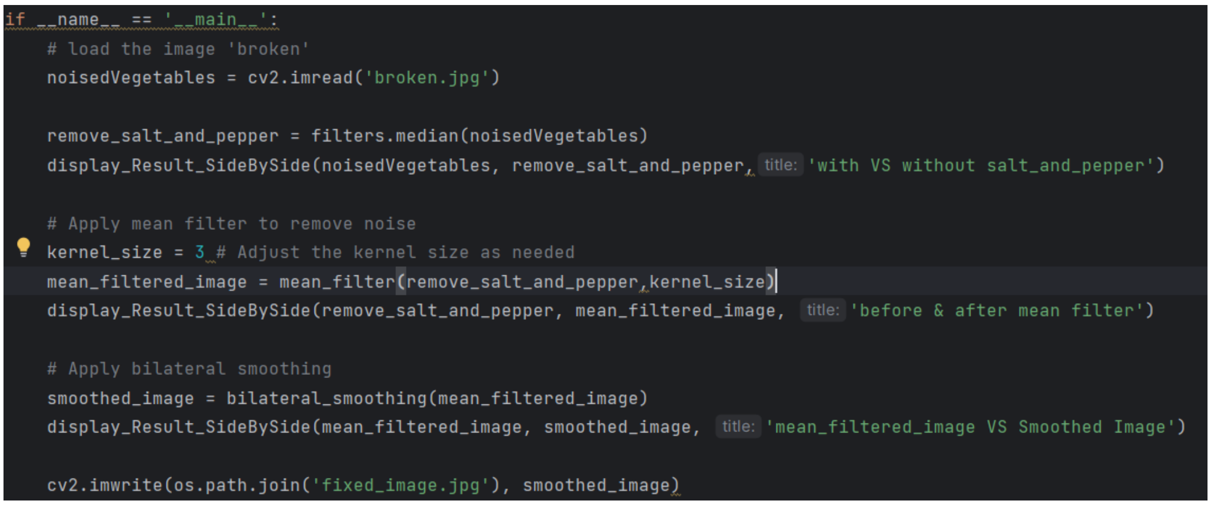
3. Subsequently, we employed bilateral smoothing to further refine the image. Bilateral smoothing is particularly effective at preserving significant edges while simultaneously reducing noise.

mean_filtered_image VS Smoothed Image



By employing these techniques in sequence, we aimed to restore the image to its optimal quality while minimizing disruption caused by noise.

Below is our code:

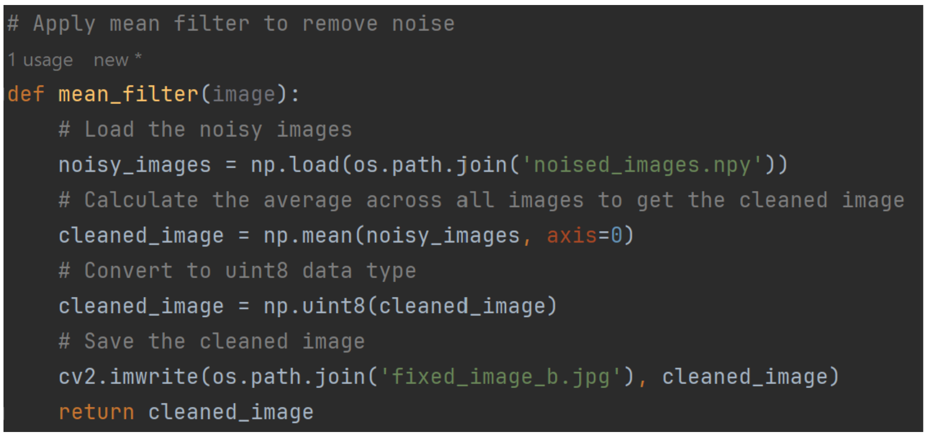


```
if __name__ == '__main__':
    # load the image 'broken'
    noisedVegetables = cv2.imread('broken.jpg')

    remove_salt_and_pepper = filters.median(noisedVegetables)
    display_Result_SideBySide(noisedVegetables, remove_salt_and_pepper, title: 'with VS without salt_and_pepper')

    # Apply mean filter to remove noise
    kernel_size = 3 # Adjust the kernel size as needed
    mean_filtered_image = mean_filter(remove_salt_and_pepper,kernel_size)
    display_Result_SideBySide(remove_salt_and_pepper, mean_filtered_image,  title: 'before & after mean filter')

    # Apply bilateral smoothing
    smoothed_image = bilateral_smoothing(mean_filtered_image)
    display_Result_SideBySide(mean_filtered_image, smoothed_image,  title: 'mean_filtered_image VS Smoothed Image')

    cv2.imwrite(os.path.join('fixed_image.jpg'), smoothed_image)
```

b. To enhance the original "broken" image, we utilized a collection of 200 noisy images stored in the noised_images.npy file. Upon examination, we observed that all these images are variations of the original image with random instances of salt-and-pepper noise. This insight led us to hypothesize that among these images, the correctly colored pixels for most of the salt-and-pepper noise instances could be discerned.

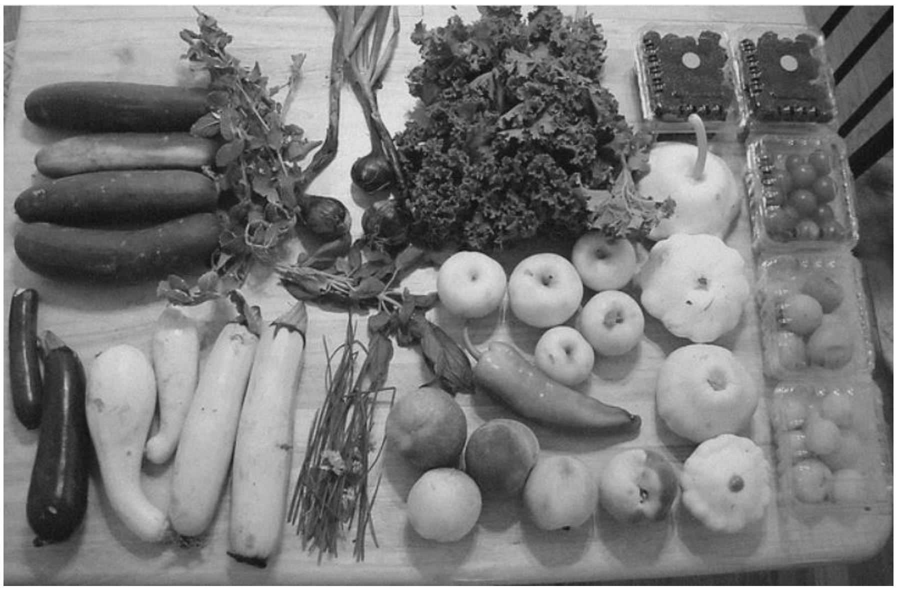
We adopted a method wherein we generated a median image from the collection of 200 noisy images. By computing the median pixel value for each pixel position across all images, we effectively reduced the noise while preserving the original details of the image. This approach leverages the fact that the median operation is robust to outliers, making it suitable for noise reduction tasks like this.

Below is our code:



```
# Apply mean filter to remove noise
1 usage   new *
def mean_filter(image):
    # Load the noisy images
    noisy_images = np.load(os.path.join('noised_images.npy'))
    # Calculate the average across all images to get the cleaned image
    cleaned_image = np.mean(noisy_images, axis=0)
    # Convert to uint8 data type
    cleaned_image = np.uint8(cleaned_image)
    # Save the cleaned image
    cv2.imwrite(os.path.join('fixed_image_b.jpg'), cleaned_image)
    return cleaned_image
```

And this is the final result:



As you can see, the resulting fixed image exhibits a significant reduction in noise while retaining the essential details of the original image, providing a visually improved version for analysis and interpretation.