# Homework 1: Image Enhancement Using Spatial Filters

## Part I. Implementation (5%)

### Padding

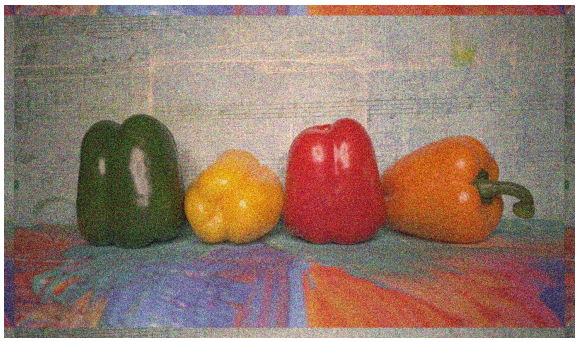
```python
def padding(input_img, kernel_size):
    ############### YOUR CODE STARTS HERE ###############
    psize = kernel_size // 2
    method_map = {0: "constant", 1: "edge", 2: "wrap", 3: "reflect"}

    if PADDING_METHOD == 0:
        # 0: Zero padding
        B = np.pad(input_img[:, :, 0], pad_width=psize, mode='constant', constant_values=0)
        G = np.pad(input_img[:, :, 1], pad_width=psize, mode='constant', constant_values=0)
        R = np.pad(input_img[:, :, 2], pad_width=psize, mode='constant', constant_values=0)
    else:
        # 1: Clamp padding
        # 2: Wrap padding
        # 3: Mirror Padding
        B = np.pad(input_img[:, :, 0], pad_width=psize, mode=method_map[PADDING_METHOD])
        G = np.pad(input_img[:, :, 1], pad_width=psize, mode=method_map[PADDING_METHOD])
        R = np.pad(input_img[:, :, 2], pad_width=psize, mode=method_map[PADDING_METHOD])
    output_img = np.dstack((B, G, R))
    ############### YOUR CODE ENDS HERE ################
    return output_img
```

To ensure that the output size of the convolution operation matches the input size, we use `padding()`. During class, we introduced four different padding methods, all of which can be implemented easily using `np.pad()` by adjusting the `mode` attribute. The padded images for each method are shown below.

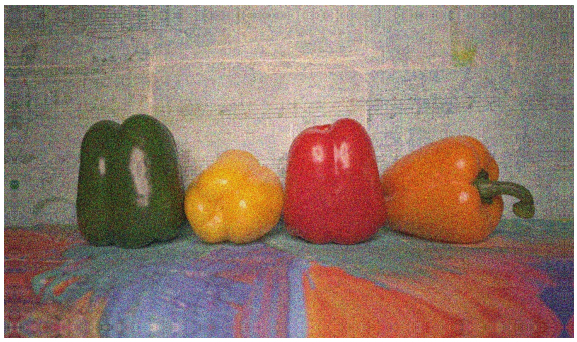
Since the original input image is a color image, padding needs to be applied separately to each channel. After padding each channel individually, we can combine them using `np.dstack()` to create the final padded image.

> Zero padding

> Clamp padding



> Wrap padding



> Mirror padding

## Convolution

```python
def convolution(input_img, kernel):
    ############### YOUR CODE STARTS HERE ###############
    input_img.astype(np.float32)
    output_img = np.zeros_like(input_img, dtype=np.float32)
    ksize = len(kernel)
    if CONV_METHOD == 0:
        padded_img = padding(input_img, ksize)
        for i in range(input_img.shape[0]):
            for j in range(input_img.shape[1]):
                for channel in range(input_img.shape[2]):
                    output_img[i, j, channel] = np.sum(padded_img[i:i+ksize, j:j+ksize, channel] * kernel)
    else:
        # Ensure the kernel is the same size as the input image
        padded_kernel = np.zeros_like(input_img[:, :, 0], dtype=np.float32)
        padded_kernel[:ksize, :ksize] = kernel

        # Perform FFT on both the input image and the padded kernel for each channel
        kernel_fft = np.fft.fft2(padded_kernel)
        for channel in range(input_img.shape[2]):
            input_fft = np.fft.fft2(input_img[:, :, channel])
            output_fft = input_fft * kernel_fft
            output_img[:, :, channel] = np.real(np.fft.ifft2(output_fft))

    output_img.astype(np.uint8)
    ############### YOUR CODE ENDS HERE ################
    return output_img
```

This function implements a 2D convolution operation that applies a kernel (filter) to an input image, supporting two convolution methods: **direct convolution** and **frequency domain convolution**.

- **Direct Convolution**: Applies the kernel directly in the spatial domain.
- **FFT-based Convolution**: Uses the Fourier transform to perform convolution in the frequency domain, which can be faster for large images and kernels.

Before performing the convolution, the input image is first converted from an unsigned integer type to `float32` to ensure accurate calculations. An `output_img` array, initialized with zeros and matching the shape of `input_img`, is created to store the convolution results.

1. **Spatial Domain Convolution (Direct Convolution)**

   The input image is padded to handle boundary conditions using the `padding()` function introduced earlier.

   For each pixel in each channel of the original input image, a window of size `ksize x ksize` is extracted from the padded image. This window is then element-wise multiplied with the `kernel`, and the sum of the resulting values is computed using `np.sum()` to produce the final convolution result for that pixel.

   The result is stored in the `output_img` array.

2. **Frequency Domain Convolution (FFT)**

   A padded version of the kernel, sized to match the input image dimensions (excluding channels), is created. Initially filled with zeros, this padded kernel has the original kernel placed in the top-left corner, effectively padding the kernel to match the input size.

   FFT is then applied to the padded kernel and each channel of the input image using `np.fft.fft2()`. The convolution result in the frequency domain, `output_fft`, is obtained through element-wise multiplication of `input_fft` and `kernel_fft`. Finally, inverse FFT (`ifft2`) is applied to `output_fft` to bring the result back to the spatial domain, and only the real part is retained.

## Gaussian Filter

```python
def gaussian_filter(input_img, ksize=5, sigma=1):
    ############### YOUR CODE STARTS HERE ###############
    kernel = np.ndarray((ksize, ksize))
    center = (ksize - 1) / 2
    counter = 0
    for i in range(ksize):
        for j in range(ksize):
            x, y  = i - center, j - center
            kernel[i, j] = 1 / (2 * np.pi * sigma ** 2) * np.pow(np.e, -(x ** 2 + y ** 2) / (2 * sigma ** 2))
            counter += kernel[i, j]
    # Normalize the kernel
    kernel /= counter
    ############### YOUR CODE ENDS HERE ################
    return convolution(input_img, kernel)
```

After completing the padding and convolution operations, we can start implementing the Gaussian filter from scratch. First, we initialize the kernel as an `np.ndarray` with shape `(ksize, ksize)`. For each element `(i, j)` in the Gaussian kernel, the value is calculated using the Gaussian function:

$$g(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(\frac{-(x^2 + y^2)}{2\sigma^2}\right)$$

where `(x, y)` is computed as `(i - center, j - center)`, and `center` indicates the center position of the kernel. For example, when `ksize = 3`, the `(x, y)` coordinates for each element will be as follows:

$$\begin{bmatrix} (-1,-1) & (0,-1) & (1,-1) \\ (-1,0) & (0,0) & (1,0) \\ (-1,1) & (0,1) & (1,1) \end{bmatrix}$$

After calculating the values for each element, the final step is to normalize the kernel. This can be done by dividing each element by the sum of all elements in the kernel (referred to as `counter` in the code). This normalization ensures that the total weight of the kernel is 1, maintaining the brightness of the filtered image.

Once the Gaussian kernel of the specified `ksize` is obtained, we apply it to the input image by performing a convolution operation, and the resulting image is returned.

## Median Filter

```python
def median_filter(input_img):
    ############### YOUR CODE STARTS HERE ###############
    padded_img = padding(input_img, KSIZE)
    output_img = np.zeros_like(input_img)
    for i in range(input_img.shape[0]):
        for j in range(input_img.shape[1]):
            for channel in range(input_img.shape[2]):
                output_img[i, j, channel] = np.median(padded_img[i:i+KSIZE, j:j+KSIZE, channel])
    ############### YOUR CODE ENDS HERE ################
    return output_img
```

The median filter is a non-linear filter, so we cannot simply use a kernel and convolution to achieve the result. Instead, for each pixel in the image, we extract a neighborhood of fixed kernel size and set the median value of this neighborhood as the new value in the output image, as demonstrated in the code above. Here, I apply the median calculation separately for each color channel.

It's also important to mention that we need to pad the original image before applying the median filter; otherwise, the output image will not have the same dimensions as the input image.

### Laplacian Sharpening

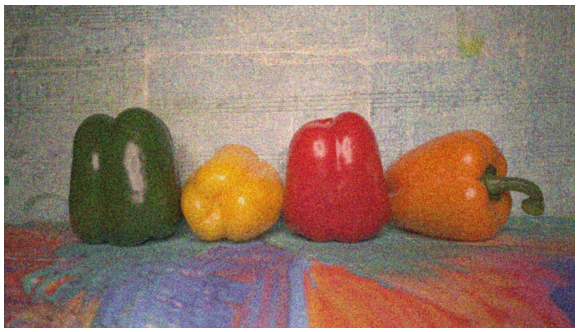
```python
def laplacian_sharpening(input_img):
    ################ YOUR CODE STARTS HERE ###############
    FILTER = 1
    if FILTER == 1:
        kernel = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
    else:
        kernel = np.array([[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]])
    ################ YOUR CODE ENDS HERE #################
    return convolution(input_img, kernel)
```

This part is relatively straightforward: we use the `FILTER` parameter to select which filter to apply. Then, we return the result of the convolution between the input image and the chosen Laplacian kernel.

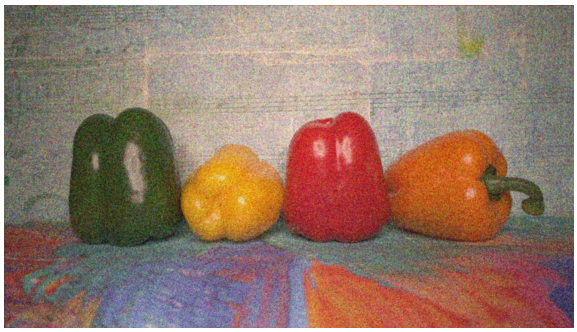
## Part II. Results & Analysis (10%):

### Gaussian filter - Different σ comparison results

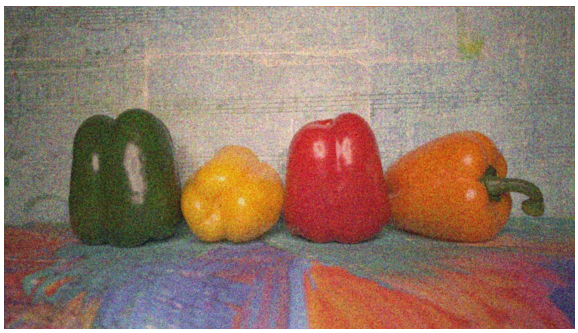
First, I set the kernel size to 3 and experimented with 4 different values for sigma. The results are shown below:



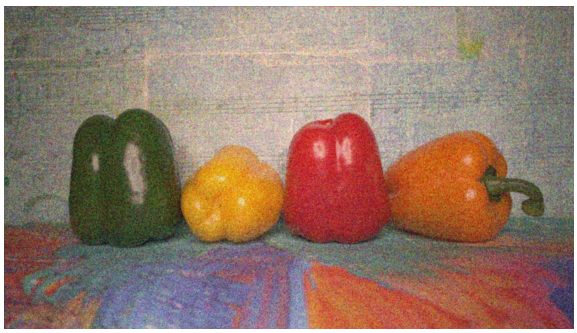
- $\sigma = 1$



- $\sigma = 3$



- $\sigma = 5$

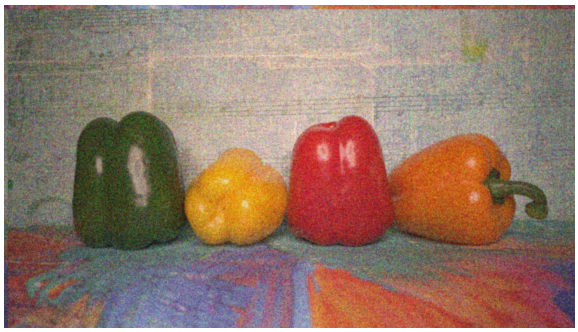


- $\sigma = 7$

As we can see, the differences are not very noticeable. When the kernel size is small, such as

$3 \times 3$, the effect of the standard deviation $\sigma$ (which controls the spread of values in a Gaussian kernel) is limited because there are too few pixels to capture the full range of values that a Gaussian distribution would typically cover. With only a few values in the kernel, changes in $\sigma$ lead to only slight adjustments in the weights, so the kernel remains relatively uniform.

So let's adjust the kernel size to 20 and see what we get.



- $\sigma = 1$

- $\sigma = 3$

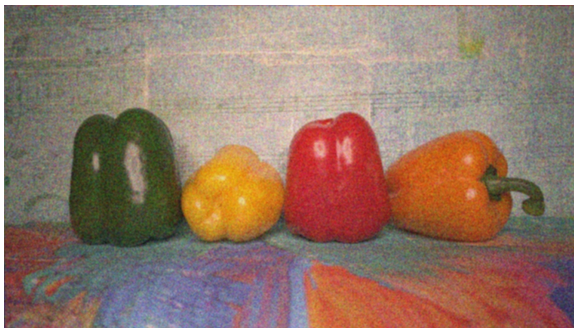
- $\sigma = 5$
- $\sigma = 7$

As we can see, with larger kernels, the Gaussian distribution can be represented more accurately, allowing for more noticeable differences in smoothing or blurring effects as $\sigma$ changes.

## Gaussian filter - Different filter size comparison  results

When using Gaussian filters of different sizes, the resulting image blur varies significantly. The result is shown below (fixed $\sigma$ to 3):

> `ksize = 3`



> `ksize = 5`

> `ksize = 7`

> `ksize = 10`

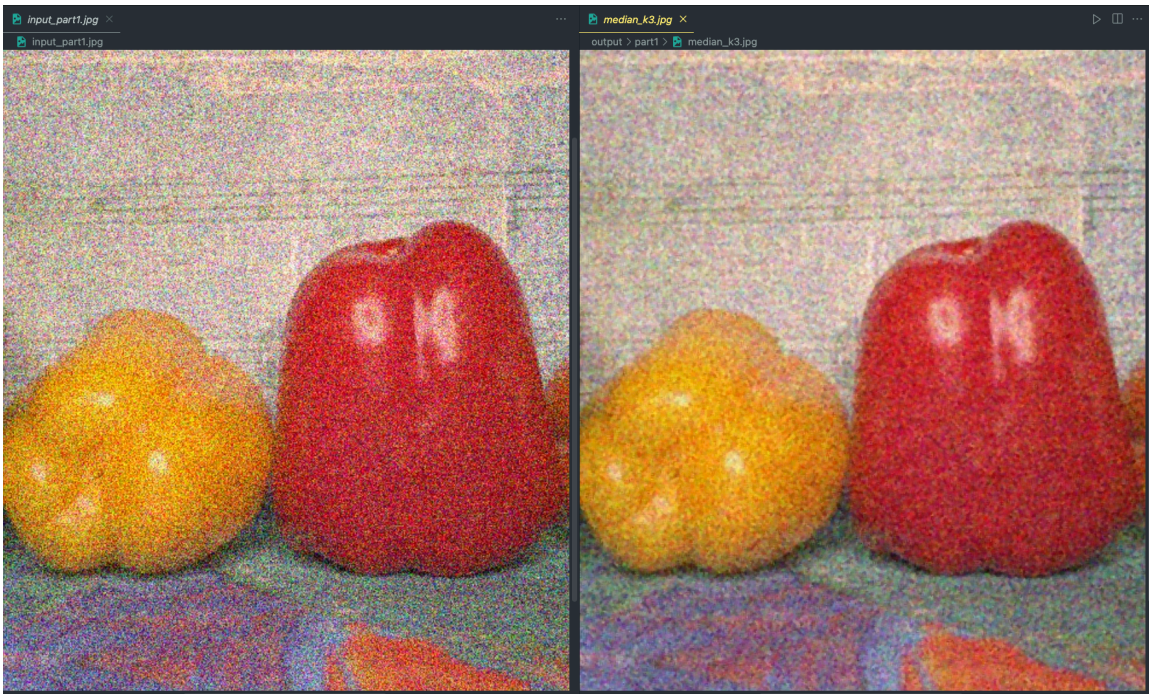
As we can see, the larger the Gaussian filter, the greater the blurring effect, with smaller filters preserving details and larger filters creating a softer, more uniform appearance. Adjusting the filter size allows for control over the balance between noise reduction and detail preservation in image processing.

**Summary**

In essence, **sigma affects the distribution's spread within a given kernel**, while **ksize changes the overall reach of the filter**. Typically, both are adjusted together to maintain the Gaussian shape, with sigma increasing as kernel size grows.

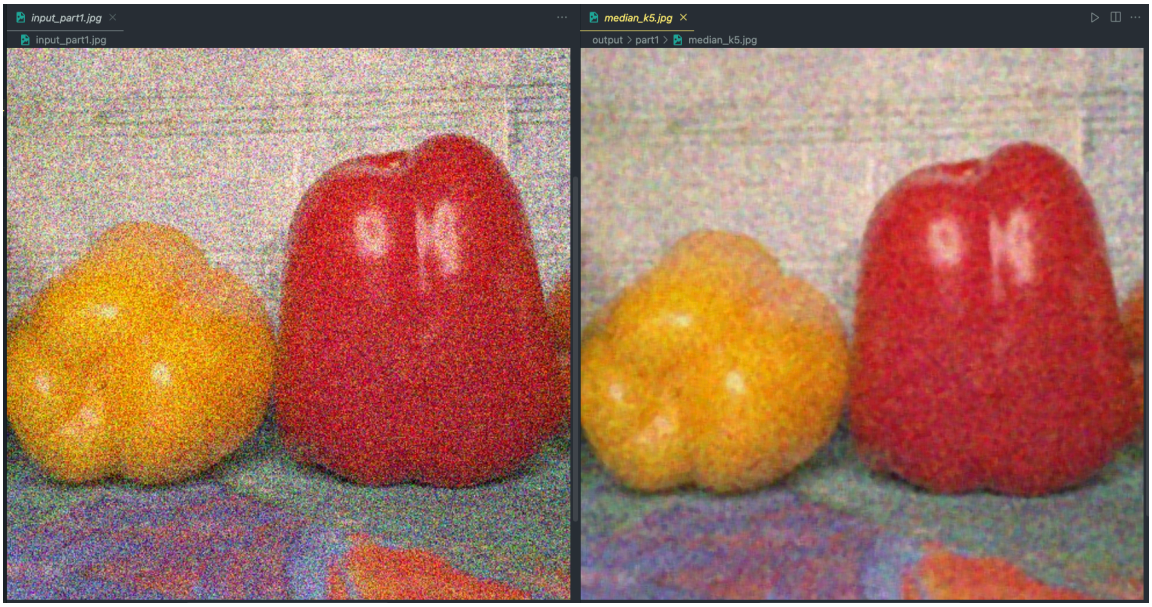
## Median filter

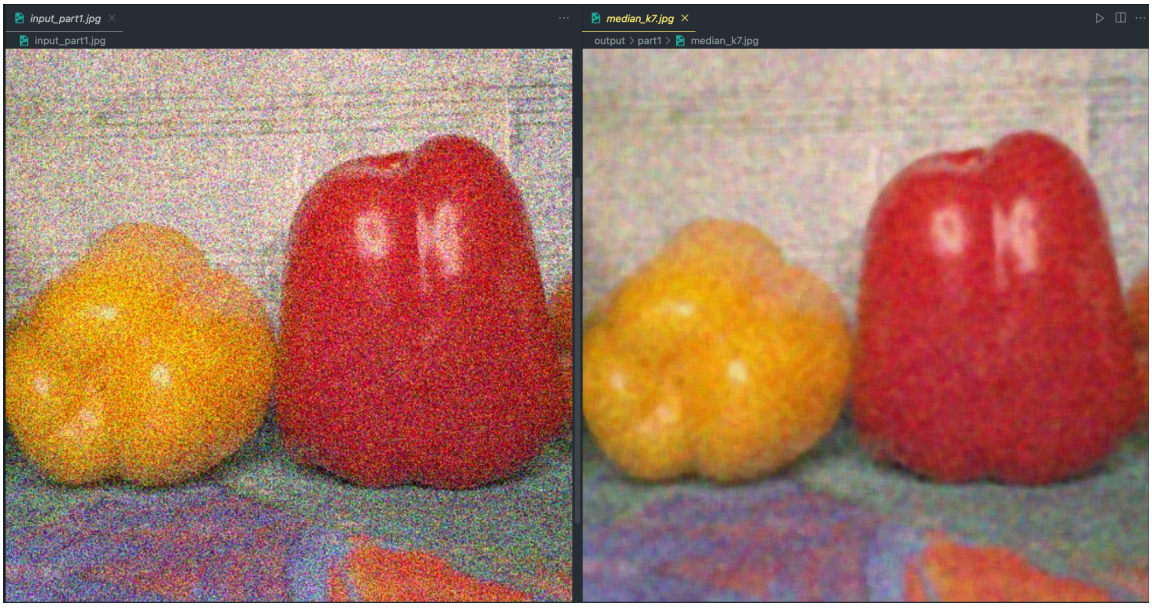
1. **Small Kernel ($3 \times 3$)**:




   - **Effect**: Minimal smoothing, primarily removing small noise specks while preserving edges and details.
   - **Result**: Most of the fine texture and details are retained, making this filter size suitable for minor noise reduction without compromising image clarity.

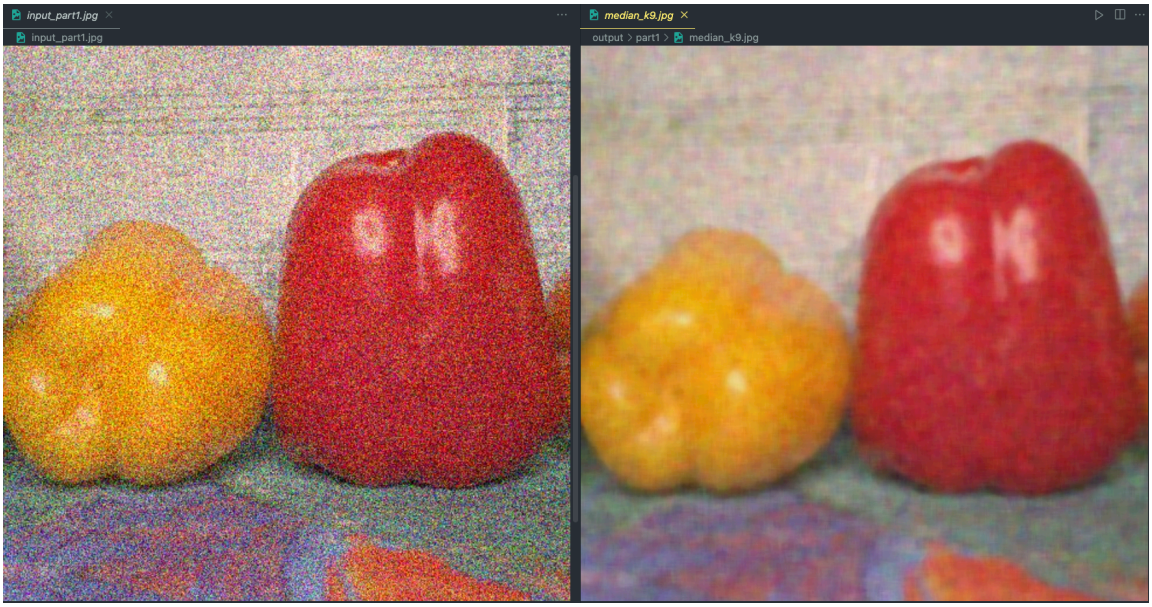
2. **Medium Kernel ($5 \times 5, 7 \times 7$)**:



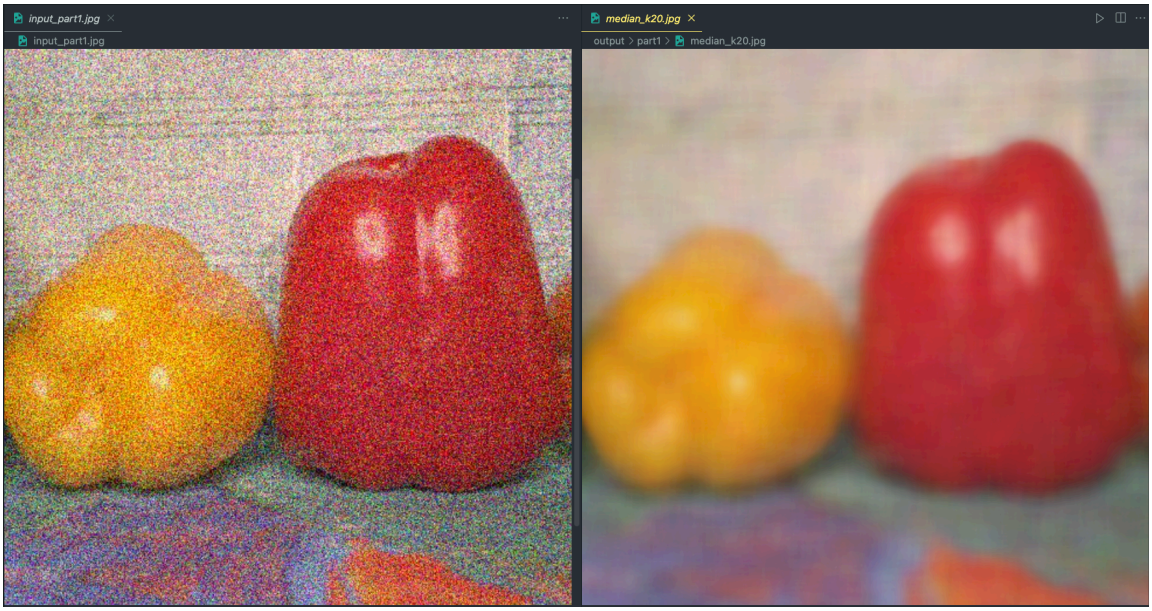

- **Effect**: Noticeable smoothing, reducing medium-level noise and beginning to blur edges slightly.
- **Result**: Fine details start to fade, but the image retains recognizable edges. These sizes effectively reduce noise while maintaining structural integrity.

3. **Larger Kernel ($9 \times 9$)**:




- **Effect**: Significant noise reduction with more prominent smoothing. Larger areas are averaged, resulting in a more pronounced blurring effect.
- **Result**: More details are lost, especially in high-frequency areas, but the image appears cleaner. This size is useful for heavier noise reduction where detail retention is less critical.

4. **Very Large Kernel (**$20 \times 20$**)**:




- **Effect**: Extreme blurring, heavily smoothing the entire image and erasing most fine details and edges.

- **Result**: The image looks soft and lacks definition. Such a large kernel size results in a median value that is overly averaged, making the image look almost out-of-focus.

In summary:

- **Smaller kernels** are best for retaining detail while removing minor noise.
- **Medium kernels** provide a balance between noise reduction and detail retention.
- **Larger kernels** focus on noise reduction at the expense of details, leading to a cleaner but softer image.

## Smoothing Spatial Filters Comparison

The **Gaussian filter** and **median filter** both are used for noise reduction, but they differ significantly in their effects and applications:

1. **Effect on Edges**
   - **Gaussian Filter**: Tends to blur edges along with noise. This is because the Gaussian filter applies a weighted average, which inherently smooths sharp transitions and edges in the image.
   - **Median Filter**: Preserves edges better than the Gaussian filter. Since it only changes pixels to match nearby values rather than averaging them, edges remain sharper. This makes the median filter effective for noise types where preserving edges is important.
2. **Effect on Different Noise Types**
   - **Gaussian Filter**: Best for **Gaussian noise (the noise type of `input_part1`)**. It provides a balanced smoothing effect over the entire image but may not be effective for impulse noise (salt-and-pepper).
   - **Median Filter**: Ideal for **impulse noise** (salt-and-pepper noise) because it directly removes small outliers without averaging. It's less effective for Gaussian noise, as it doesn't consider a pixel's intensity in its filtering process.
3. **Processing Differences**
   - **Gaussian Filter**: A convolution-based filter, computationally efficient with fast implementations available, especially using FFT (Fast Fourier Transform).

- **Median Filter**: Non-linear and more computationally intensive because it involves sorting the neighborhood pixels to find the median value.

**Summary of Visual Differences**

- **Gaussian Filter Results**: Smoother image, with noise and fine details gradually blurred, including edges.
- **Median Filter Results**: Noise is reduced, edges are preserved, and the image looks less blurred. Suitable for noise removal in images where edges are critical, like text or object boundaries.

## Laplacian Filter Comparison

> Filter 1 result

> Filter 2 result

1. **Filter 1:**
   - **Kernel**:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

   - This filter emphasizes edges moderately. Since it only increases the center pixel by 5 and reduces neighboring pixels by -1, it enhances edges without over-sharpening or producing strong artifacts. It provides a subtle sharpening effect while preserving more of the original image's details.
   - **Result**: The image looks moderately sharpened. The edges of the birds, grass, and other objects are clearer, but the sharpening is controlled, preserving a more natural look.

2. **Filter 2**

   - **Kernel**:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

   - **Effect**: This filter is a stronger edge enhancer, as it increases the center pixel value by 9 and decreases all neighboring pixels by -1. This results in a more pronounced sharpening effect, which may create stronger edges and even exaggerate them, possibly introducing artifacts or noise, especially around high-contrast areas.
   - **Result**: The image appears significantly sharpened, with more dramatic contrast around edges. This filter enhances the details much more aggressively, potentially making edges overly prominent and possibly introducing a slightly "haloed" or more artificial look.

# Part III. Answer the questions (15%):

1. **Please describe a problem you encountered and how you solved it.**

   While testing and comparing different filters, I found that **convolution filters were running extremely slowly**, especially on larger kernel sizes. This is because convolution in the spatial domain requires multiple nested loops, making it computationally intensive. To address this, I decided to use **FFT (Fast Fourier Transform)** to speed up the convolution process. By transforming both the image and the kernel into the frequency domain, I could replace convolution with element-wise multiplication, which significantly reduced the computation time and improved the filter's efficiency.

2. **What padding method do you use, and does it have any disadvantages? If so, please suggest possible solutions to address them.**

   I implemented four padding methods: **Zero padding**, **Clamp padding**, **Wrap padding**, and **Mirror padding**. Most of my testing relied on **Zero padding** for simplicity. However, Zero padding introduces black borders around the edges of the image, which can create unnatural effects near the boundaries, especially with larger filters. This border can lead to edge artifacts where the transition between the original image and the zero-padded area is abrupt.

To mitigate this, **Mirror padding** or **Clamp padding** could be better alternatives. Mirror padding creates a smoother transition by reflecting the edge pixels, reducing artifacts while preserving natural edges.

3. **What problems do you encounter when using Gaussian filter and median filter to denoise images? Please suggest possible solutions to address them.**

   Initially, when applying the **Gaussian filter**, the output did not display correctly. I discovered that this was due to data type issues; since Gaussian filters involve floating-point calculations, I needed to convert the input image to `float32` before processing, and then convert it back to `uint8` at the end. This resolved the display issue.

   As for the **median filter**, I found that it did not produce the expected level of noise reduction, as some noise remained visible, and edges were still blurred. This might be because I applied the median filter separately on the three RGB channels, which can distort colors and details. A possible improvement could be to **convert the image to HSV color space** first and then apply the median filter. This approach would preserve color information while reducing noise, especially in bright regions, potentially improving denoising quality.

4. What problems do you encounter when using Laplacian filters to sharpen images? Please suggest possible solutions to address them.

   One issue with using **Laplacian filters** for sharpening is that they can introduce **noise amplification** and **edge artifacts**, especially in high-frequency areas. Since the Laplacian filter enhances edges by highlighting rapid intensity changes, it also intensifies any existing noise, which can lead to a "halo" effect around edges or make the image look overly sharp and unnatural.

   To address this, a potential solution is to use **Unsharp Masking** instead. This technique involves first applying a Gaussian blur to smooth out noise and then subtracting it from the original image to create a sharpened effect. This method balances edge enhancement with noise suppression. Another approach is to use **adaptive filters** that apply different levels of sharpening based on local image characteristics, reducing sharpening in already high-contrast regions to avoid exaggerated edges.