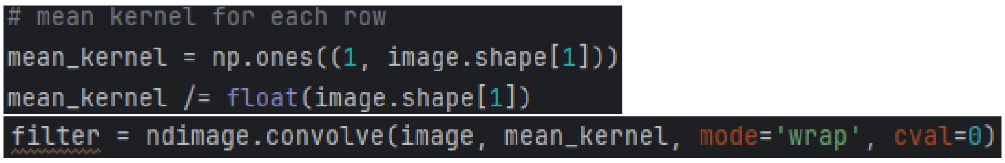
Image Processing – HW3

Problem 1

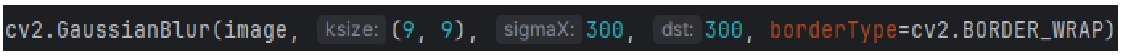
We've received an image and had to guess which nine filters were executed

1. Mean copy – after debugging and reviewing the image array, we've found that each row is the same pixel value, and it is the average of all pixels on that row. So we've found the average of each row and make a copy image keeping that value on the entire row.



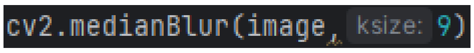
```
# mean kernel for each row
mean_kernel = np.ones((1, image.shape[1]))
mean_kernel /= float(image.shape[1])
filter = ndimage.convolve(image, mean_kernel, mode='wrap', cval=0)
```

2. Gaussian blur – the image is blurred but in a smooth way. No edges are present, so it is not bilateral. Regarding padding, we've used wrap boarder as is looks like a cyclic change.



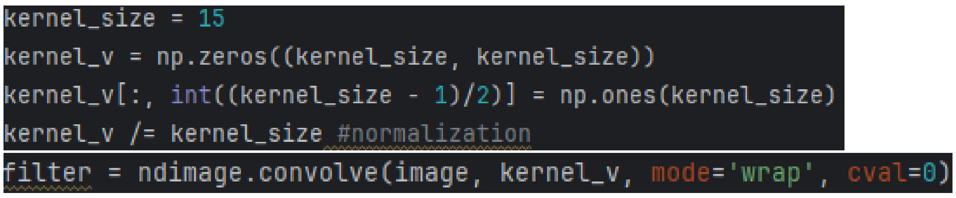
```
cv2.GaussianBlur(image, ksize: (9, 9), sigmaX: 300, dst: 300, borderType=cv2.BORDER_WRAP)
```

3. Median blur – compared to the previous filter the image is smeared, and color changes are almost binary and not smooth. This fits with median blur.



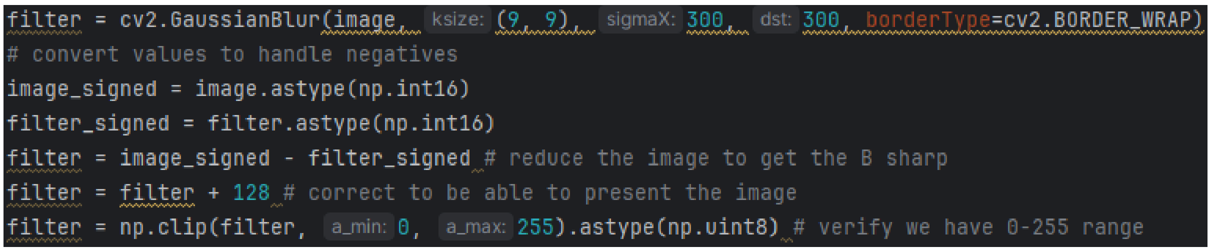
```
cv2.medianBlur(image, ksize: 9)
```

4. Motion blur – the image appears to be moving along the y axis. We took a 15x15 kernel with middle row of ones, and normalized it.



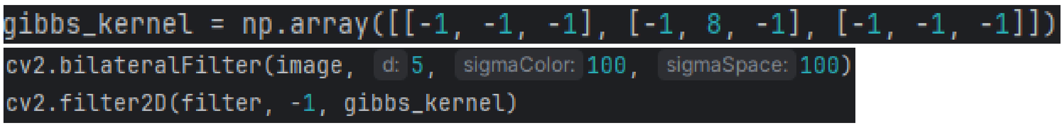
```
kernel_size = 15
kernel_v = np.zeros((kernel_size, kernel_size))
kernel_v[:, int((kernel_size - 1)/2)] = np.ones(kernel_size)
kernel_v /= kernel_size #normalization
filter = ndimage.convolve(image, kernel_v, mode='wrap', cval=0)
```

5. B sharp – as seen in the lecture, the image is the B_{sharp} which we've calculated by reducing the blurred image from the original one. The blurred image was done with gaussian filter. We've also had to treat the borders and found "wrap" was most suitable.



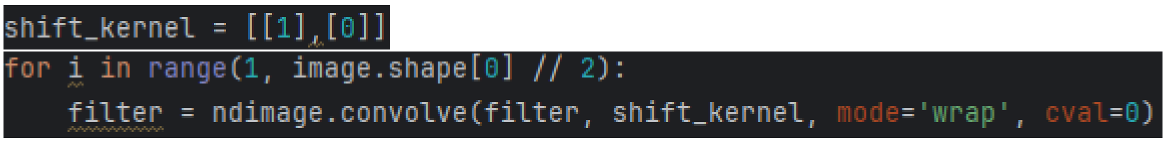
```
filter = cv2.GaussianBlur(image, ksize: (9, 9), sigmaX: 300, dst: 300, borderType=cv2.BORDER_WRAP)
# convert values to handle negatives
image_signed = image.astype(np.int16)
filter_signed = filter.astype(np.int16)
filter = image_signed - filter_signed # reduce the image to get the B sharp
filter = filter + 128 # correct to be able to present the image
filter = np.clip(filter, a_min: 0, a_max: 255).astype(np.uint8) # verify we have 0-255 range
```

6. Gibbs artifact – similar to the previous image, this is the other angle of the B_{sharp}, this time the image shows the artifact itself. Since the borders are very distinct, we've used bilateral filter to blur the image, then used the following mask for convolution.



```
gibbs_kernel = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
cv2.bilateralFilter(image, d: 5, sigmaColor: 100, sigmaSpace: 100)
cv2.filter2D(filter, -1, gibbs_kernel)
```

7. Translation – the filter is a cyclic shifting by half the height of the image – since a big kernel is heavy on computation, we've made a small kernel and ran image_height/2 times



```
shift_kernel = [[1],[0]]
for i in range(1, image.shape[0] // 2):
    filter = ndimage.convolve(filter, shift_kernel, mode='wrap', cval=0)
```

8. We believe that the filter is the identity mask, we've applied it and calculated MSE which came out zero.

```python
identity_kernel = np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]])
ndimage.convolve(image, identity_kernel, mode='wrap', cval=0)
```

9. Sharpening – the borders/edges of each object are stressed. We've used the following mask for the convolution

```python
sharpen_kernel = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
cv2.filter2D(image, -1, sharpen_kernel)
```

Problem 2

<u>Bilateral Cleaning Algorithm</u>

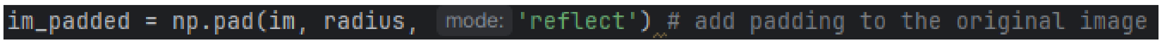
We loop on all three images and send them the *clean_Gaussian_noise_bilateral* method

Method: clean_Gaussian_noise_bilateral

Input: the input image to be corrected (array), radius of the kernel, std of intensity and of spatial (both floats)

output: corrected image (array unit8)

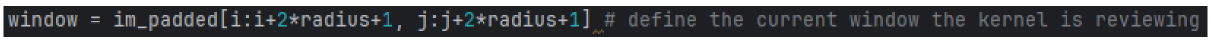
We prepare a new “empty” image of the same shape as the input image. Since we use convolution, we must save all results on a different image in order not to interfere with results between convolutions. We also padded the image in order to account for all pixels, chose boarder type “reflect” in order not to add bias in the edges



```
im_padded = np.pad(im, radius, mode: 'reflect') # add padding to the original image
```

We then prepared the spatial mask which will be constant on each iteration, with method *“create_spatial_mask”*

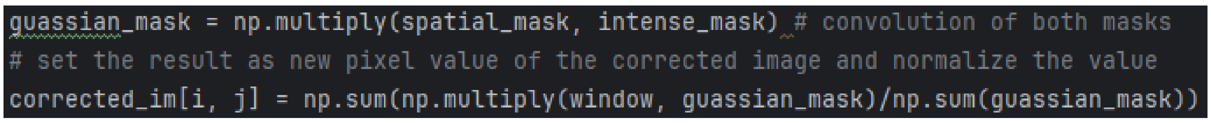
We now iterate on the entire image to calculate the new pixel value after implementing the gaussian mask. On each iteration we pass a window of the kernel size



```
window = im_padded[i:i+2*radius+1, j:j+2*radius+1] # define the current window the kernel is reviewing
```

Then calculate the intensity mask with the method *“create_intensity_mask”*

We use convolution between intensity and spatial masks, and then with the window (the order is not relevant since all arrays are the same and convolution is associative). The results are normalized and saved in the new image.



```
guassian_mask = np.multiply(spatial_mask, intense_mask) # convolution of both masks
# set the result as new pixel value of the corrected image and normalize the value
corrected_im[i, j] = np.sum(np.multiply(window, guassian_mask)/np.sum(guassian_mask))
```

Method: create_spatial_mask

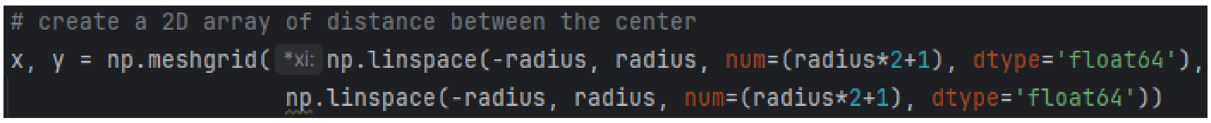
input: radius (integer), std deviation used for gaussian spatial distance (float)

output: gaussian spatial mask in float64 format

use the formula

$$g_s\,[x,y] = \exp\left(-\frac{(x-i)^2+(y-j)^2}{2\sigma^2}\right)$$

We’ve used np.meshgrid for the array



```
# create a 2D array of distance between the center
x, y = np.meshgrid( xi: np.linspace(-radius, radius, num=(radius*2+1), dtype='float64'),
                   np.linspace(-radius, radius, num=(radius*2+1), dtype='float64'))
```

Method: create_intensity_mask

input: the window to be corrected (array), std of intensity, the intensity value of the center pixel output: masked window (array)

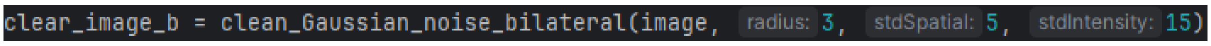
use the formula

$$g_i[x,y] = \exp\left(-\frac{(\mathrm{im}[x,y]-\mathrm{im}[i,j])^2}{2\sigma^2}\right)$$

We are now ready to try and correct each of the three images

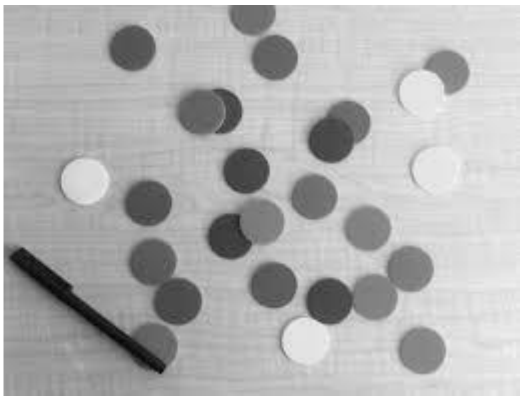
Balls picture:

In the following image the noise comes from the JPG compression. To get a smooth picture we've had to compromise on the background. But with the chosen parameters we were able to smooth the circles and keep the colors. Bilateral filter is a good fit for this task, although non local mean could also work as there are a lot of repetitions in the image.

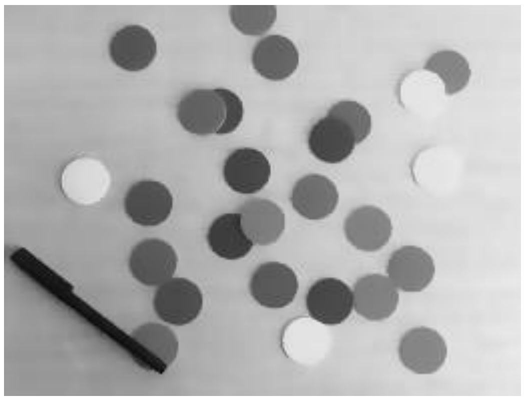


```
clear_image_b = clean_Gaussian_noise_bilateral(image, radius: 3, stdSpatial: 5, stdIntensity: 15)
```

Before Fix

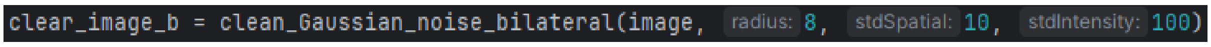


After Fix



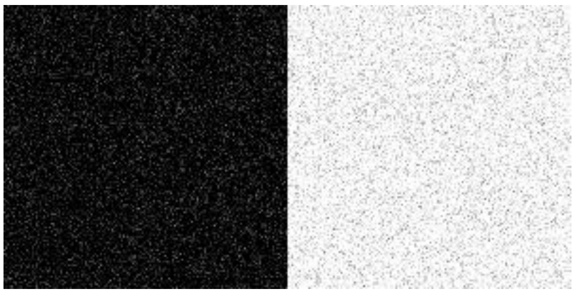
Noisy picture:

Gaussian filter is not a good fit for this image. A filter to take the median or the max will be better since when using gaussian formula the black will never be completely black as it will have an effect from white pixels. The border between the black and white will be preserved but some noise will appear in the radius of the chosen kernel. Min/max filter will be better for this image, but we've tried our best.



```
clear_image_b = clean_Gaussian_noise_bilateral(image, radius: 8, stdSpatial: 10, stdIntensity: 100)
```

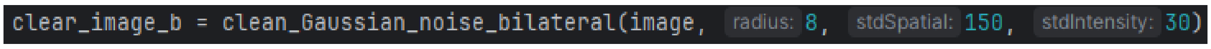
Before Fix



After Fix

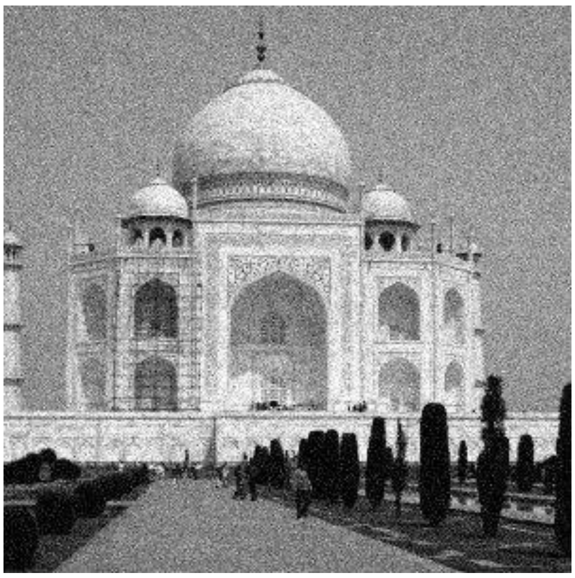
Taj picture:

This image a good case for this filter since it has many areas with very low transitions, and the edges are straight and/or distinct. We've tried to give more focus to the spatial and low intensity order not to bring too much noise from the sky/trees. The results are below

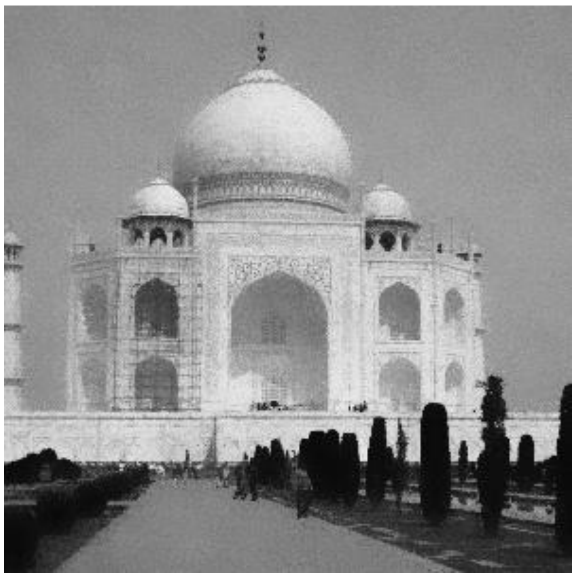


```
clear_image_b = clean_Gaussian_noise_bilateral(image, radius: 8, stdSpatial: 150, stdIntensity: 30)
```

Before Fix



After Fix

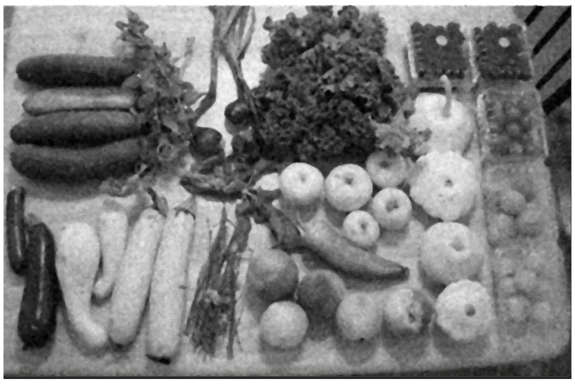


Problem 3

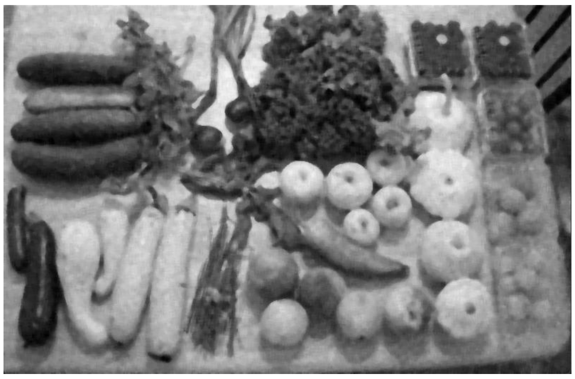
<u>Section a</u>

We kept in mind that the best filter for salt and pepper noise is median filter. Trying to apply it removed most of the details and the image was still noisy

K=5

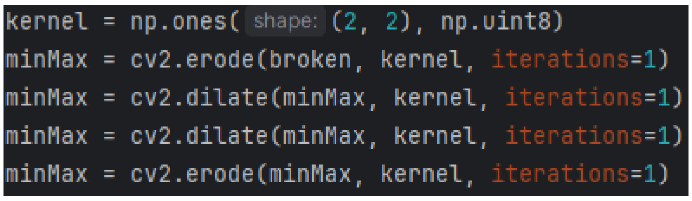


K=7



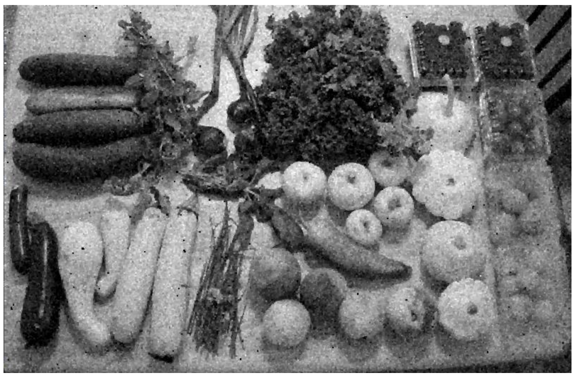
So we've decided to implement two filters right after the other and see if we'll get a better outcome. We've tried min max filter first

Min max code, as it is not in the CV library



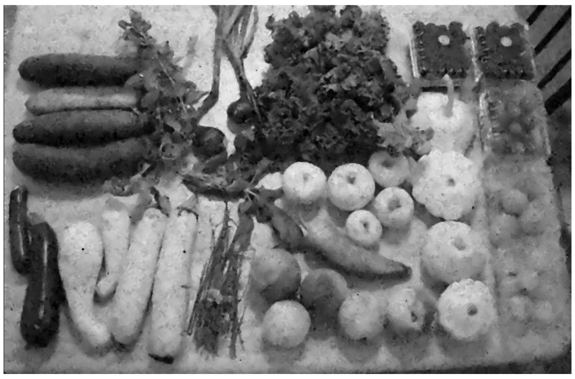
```
kernel = np.ones( shape: (2, 2), np.uint8)
minMax = cv2.erode(broken, kernel, iterations=1)
minMax = cv2.dilate(minMax, kernel, iterations=1)
minMax = cv2.dilate(minMax, kernel, iterations=1)
minMax = cv2.erode(minMax, kernel, iterations=1)
```

This method was able to remove most of the noise and still keeping most of the details.

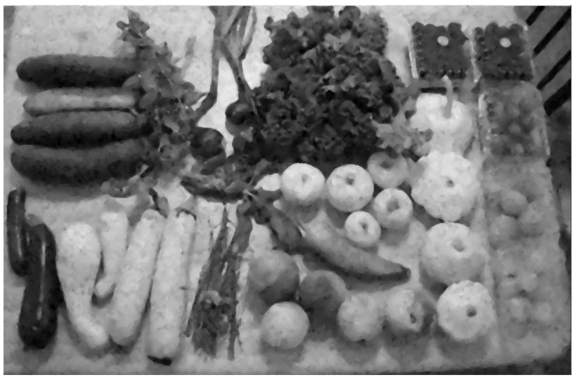


We've applied median filter as a second step but couldn't remove the noise without a lot of compromise on the details.

K=3

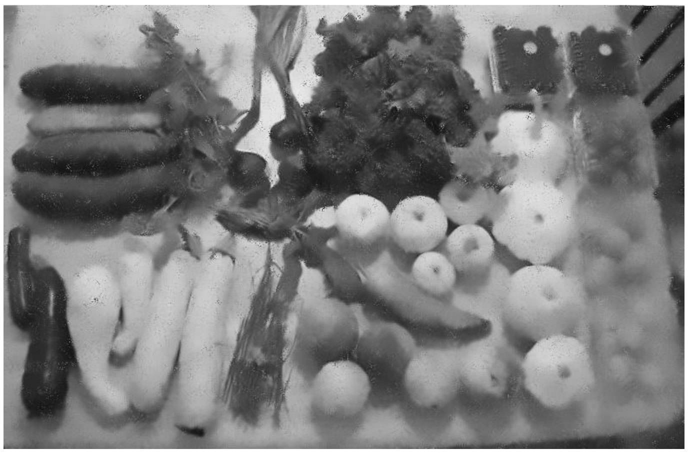


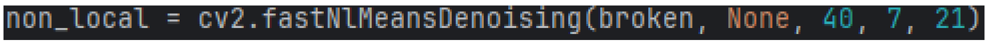
K=5



Our second approach was to use non-local means

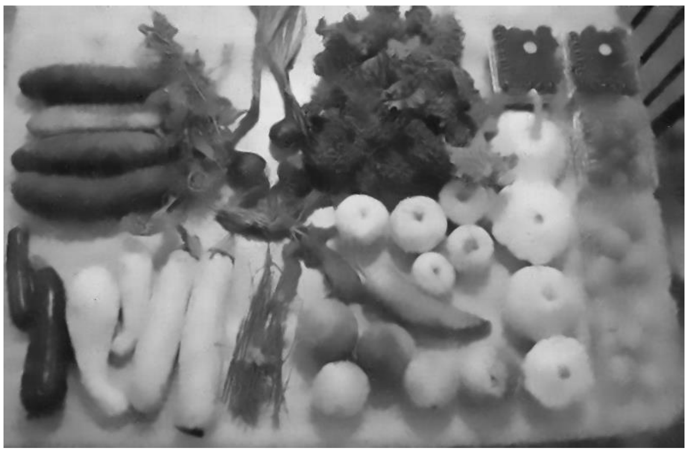
Non-local means was reviewed in the lecture and approved with Alon during the Tirgul





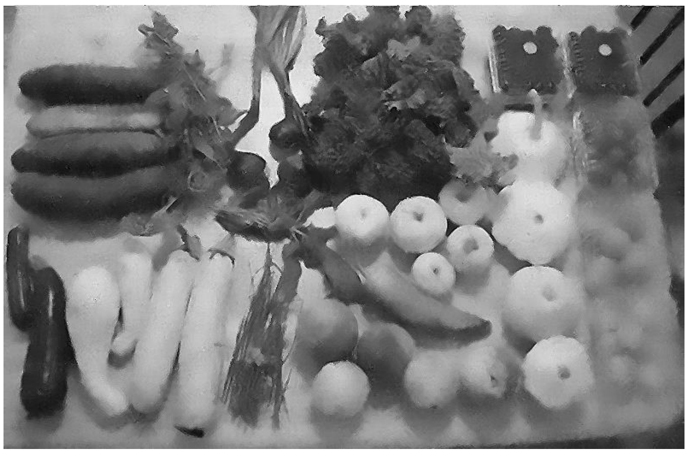
```
non_local = cv2.fastNlMeansDenoising(broken, None, 40, 7, 21)
```

Results were good. Most noise was removed and still most details are present. Then median blur with k=5 which was not a good fit with min/max method, performed much better after local mean. We did have to compromise on the bottom right corner, but we could still see the original objects

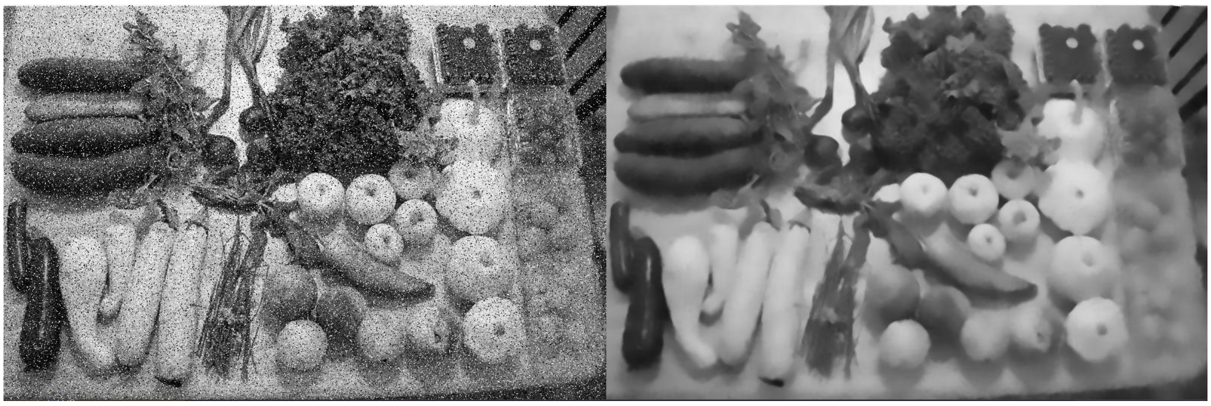


We've tried to sharpen the image with sharpening kernel which to bring back some of the details and to differentiate between the objects

0	-1	0
-1	5	-1
0	-1	0

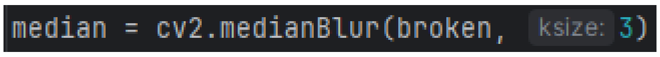


Results comparing original and our final image are below. Not perfect, but no noise and many of the fine details are present.

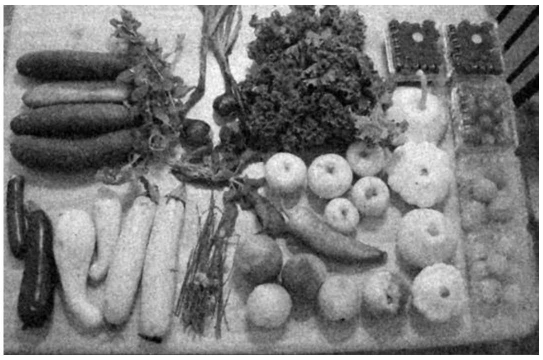


The last option we've tried is first a median filter, and then bilateral filter

After median filter, we tried to keep the kernel as small as possible. Most salt and pepper was removed

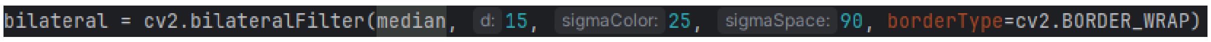


```
median = cv2.medianBlur(broken, ksize: 3)
```

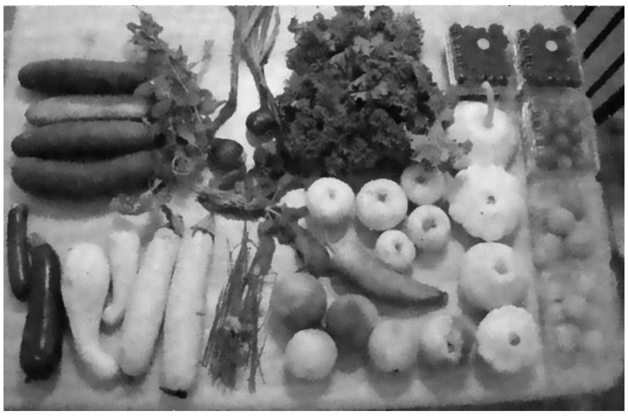


With median blur and bilateral filter we've tried to keep the kernel as small as possible due to the high noise which covered a big portion of the image

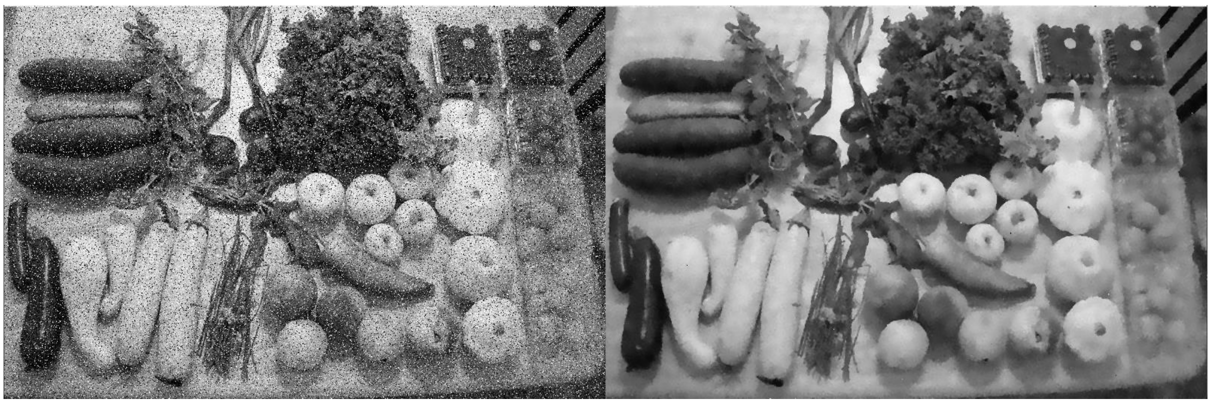
After bilateral filter



```
bilateral = cv2.bilateralFilter(median, d: 15, sigmaColor: 25, sigmaSpace: 90, borderType=cv2.BORDER_WRAP)
```



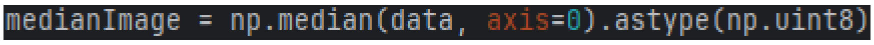
Final result showing bilateral and median blur compare to the original image is below



To sum up this first section. Using min/max did not work. Both of the two other options are acceptable on our account, it's a matter of what is more important. Using non local means keeps more clarity but compromises on loss of data towards the edge of the image, while bilateral keeps more data but looks more pixeled. On both instances we use median blur as the preferred method to clear salt and pepper noise.

Section b

We’ve loaded all images (200 of them) and took the median of each position, thinking that since they are all noisy but in most, the same pixel we’ll look at will be with the original value. Results are below which removed all noise.



```
medianImage = np.median(data, axis=0).astype(np.uint8)
```