Assignment 5

Birk Nøhr Dissing
Jakob Damgaard Olsen
Ian Pascal Møller

March 13, 2023

1. Morphology

1.1.1

In figure 1 we see the effect of applying closing and opening to the image "cells_binary_inv.png". By comparing the zoomed in sections of applying opening or closing with the original we can see features that change due to the operations. For opening we can see one effect of it in the area marked by the red circle. Here we see that the connection between the two cells have been removed. While with closing we can see that the left circle shows how two cells now are connected. But closing also has benefits such as closing holes in the cells as we can see in the marked center cell. The overall effect these two functions have on the image is the following. Opening removes stray pixels in the background and attempt to "split" cells from each other, while closing fills in cells which have holes in them, while in the process sometimes also linking cells together.

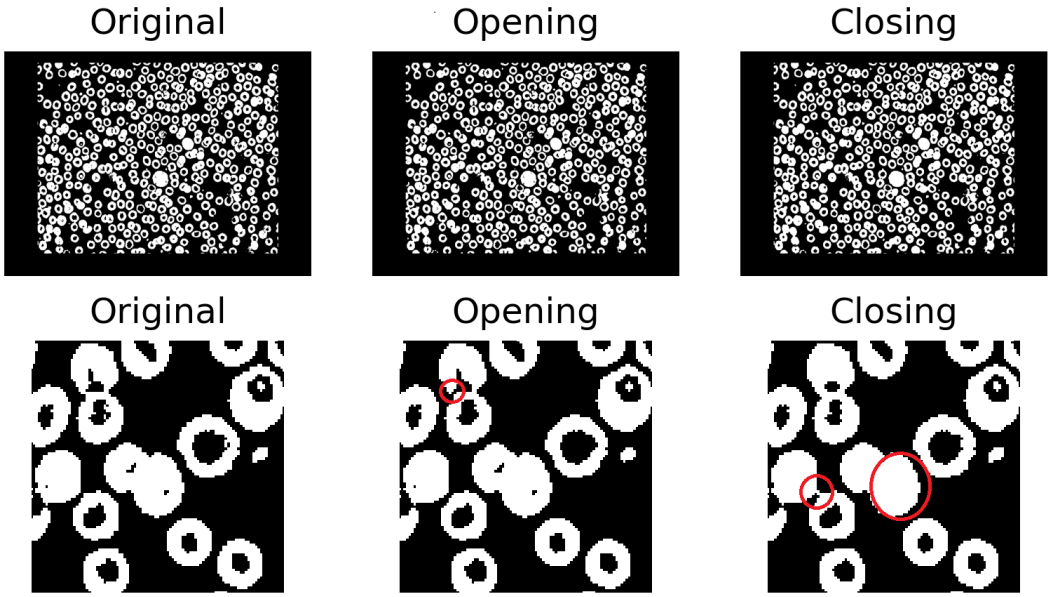

Figure 1: Opening and closing applied to "cell_binary_inv.png"

1.1.2

The reason for the two results not producing the same is due to the order of how the apply erosion and dilation. Opening is the act of performing erosion first then dilation, whereas closing performs dilation first then erosion. Therefore opening results in the removal of small objects and thin features, which are then partially recovered by dilation. Closing however starts with filling in small holes with dilation and then removing extra added structure with erosion. These aspect of the two methods is also the reason for it to be a challenge to separate the cells in "cells_binary_inv.png" by only using closing and opening, since we both want to remove small pixels, fill in cells and separate cells.

1.1.3

The code used to label the cells was the following

```
disk = morph.disk(1)
openA = morph.binary_opening(A,disk)
closedA = morph.binary_closing(A,disk)
connectedopen = sm.measure.label(openA, background=255,
   connectivity=2)
connectedclosed = sm.measure.label(closedA, background=255,
   connectivity=2)
connectedA = sm.measure.label(A, background=255,connectivity=2)
```

Figure 2 shows the result of labeling the image using opening and closing. From this we see that we identified 369 cells when using opening while we identify 343 using closing. These values gives us a range of values to estimate our amount of cells, but it is not accurate. This is due to what we also could see from the zoomed in image in figure 1. We see that a lot of the cells are connected to each other, which would result in them being labeled as the same cell, while when we do closing we have even more connected cells. Comparing the two we would say that in this case opening might have performed better at labeling cells, but a concrete answer would require a more in depth analysis. Opening would also remove small object or stray pixels which could incorrectly count towards the total amount of cells.

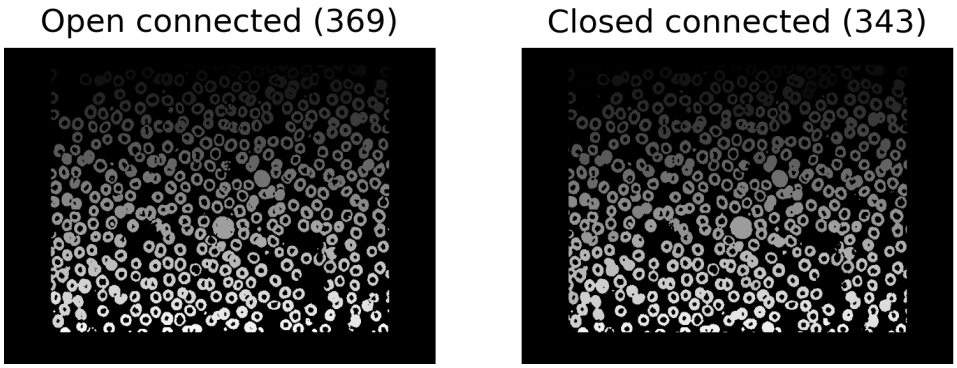

Figure 2: Effect of applying opening and closing

1.2

The method used was to do first label each coin and then compare this labeled image with one that has erosion applied to it iteratively until a coin disappeared. This also required to first make a list which had the value of the coins ordered by when they disappear, so the coin that disappears first would have its corresponding value be the first value in the list. This list was created manually by eroding the image multiple times. When a coin disappeared its value is added to the coinSum variable. Running this for 100 loops resulted in all the coins being eroded away and giving the final result of the total value of the coins on the image being $\mathbf{coinVal} = \mathbf{115.5}$. The code used to find the amount on the image was the following

```
def check_positions(m1, m2, N):
    mask = np.array(m1) == N
    result = np.any(np.array(m2)[mask] != 0)
    return result
A = imread("money_bin.jpg")
mask = A > 230
A[mask],A[~mask] = 0, 1
disk = morph.disk(5)
Atest = morph.binary_closing(A)
Atest = morph.binary_opening(Atest)
connectedA = sm.measure.label(Atest, background=255, connectivity
   =2)
coinVals,cIndex,Loops,coinSum,uq =[1,2,5,0.5,20],0,100,0,np.unique
   (connectedA)[1:]
for i in range(Loops):
    Atest,remCount = morph.binary_erosion(Atest, disk),0
    for j in uq:
        if not check_positions(connectedA, Atest, j):
            uq = uq[uq != j]
            remCount +=1
    if remCount != 0:
        coinSum += remCount*coinVals[cIndex]
        cIndex +=1
```

2. Inverse filtering

2.1

The linear shift invariant degradation model is given by

$$g = (h \cdot f) + \eta \tag{1}$$

where g is the degraded image, computed by adding a noise image or realization η to the convolution of the point spread function (PSF) h with the original image f. The implementation of this in Python can be found in 3. Examples of the output of this function can be seen in 3, where we have applied five different filters, each at four different levels of noise.

```
def ApplyLinearShiftInvariantDegradation(image, kernel, noise):
```

```
    return convolve2d(image, kernel, mode = 'same', boundary = '
        wrap') + noise
```

Listing 1: Implementation of the linear shift invariant degradation model.

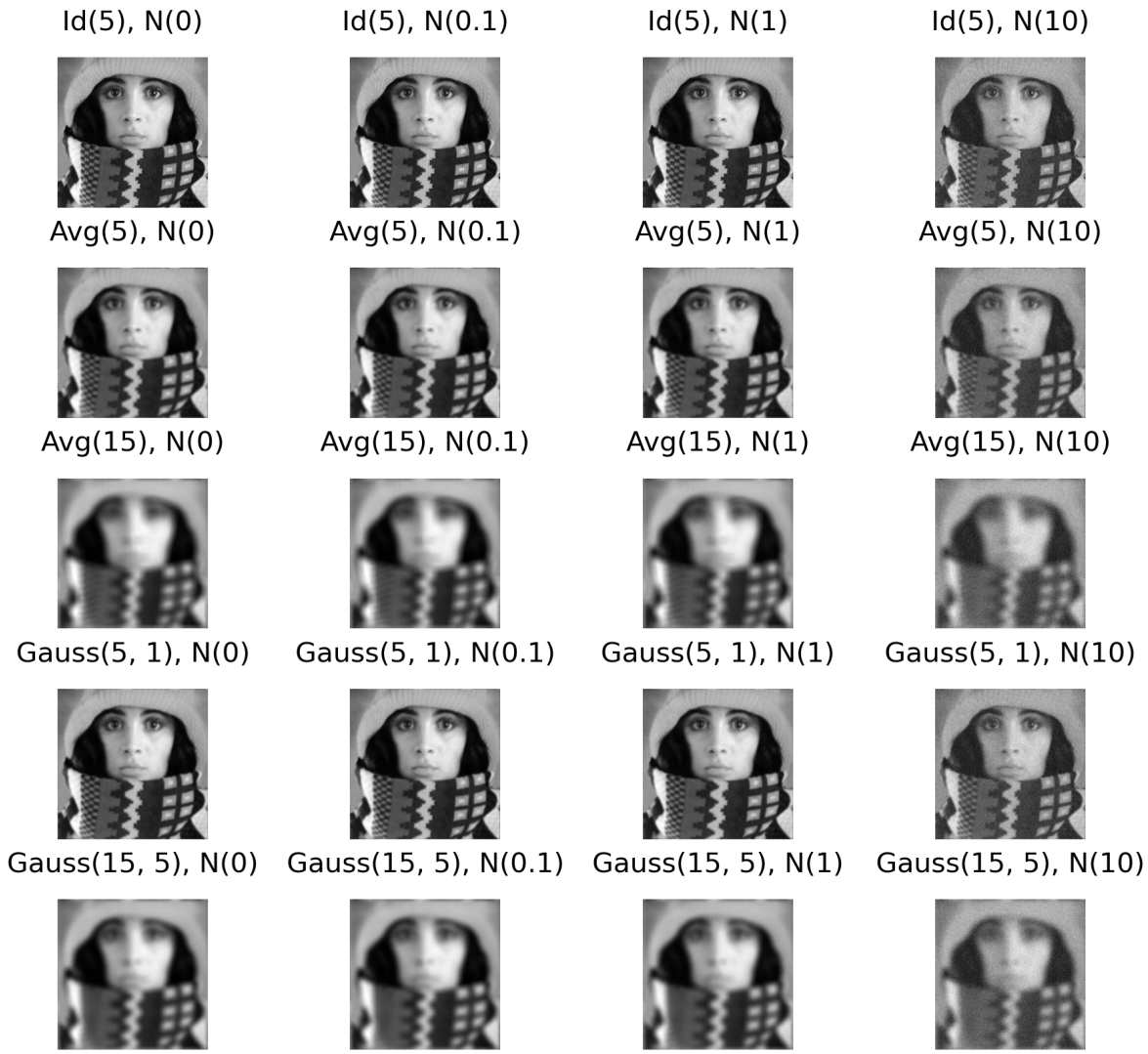

Figure 3: Examples of the linear shift invariant degradation using four different kernels and four different levels of noise. From the top: identity kernel (1 in center, 0 elsewhere); box blur/averaging kernel (1/25 everywhere in a 5x5 kernel); same as previous, but with 1/225 in a 15x15 kernel; Gaussian kernel with size 5 and width 1; Gaussian kernel with size 15 and width 5. N(x) refers to the grid of normally-distributed random noise added to the image; each pixel is added a value between 0 and x.

2.2

If we assume that no noise is present in the image, then we can theoretically recover the original image, f, as long as we know the PSF used to degrade the image. Applying the Fourier transformation to 1, we get

$$G = (H \cdot F) + N \tag{2}$$

and, if no noise present (N=0) then

$$F = \frac{G}{H} \tag{3}$$

which we can then apply the inverse Fourier transformation to in order to recover f. The implementation of this can be seen in 6, and the results of recovering the original images using the degraded images from the previous task can be found in 4. We see here that that direct inverse filtering has very mixed results, at least when considering noisy images. When there is no noise present, the images seem to be recovered quite well. However, as soon as there is any noise present, the restored image quickly begins to differ from the original: for the two smaller kernels the image can still be recognized, though they show signs of artifacting. With a moderate amount of noise or a slightly larger kernel (or both), the image fails to recover completely.

```
def ApplyDirectInverseFiltering(image, psf):
    image_ft = fft2(image)
    psf_ft = fft2(psf, image.shape)
    recovered_image = ifft2(np.divide(image_ft, psf_ft))
    return np.roll(recovered_image, shift = (2, 2), axis = (0, 1))
```

Listing 2: Implementation of the direct inverse filtering. Numpy's roll function is used to adjust the image, since it becomes offset after the Fourier transformation.

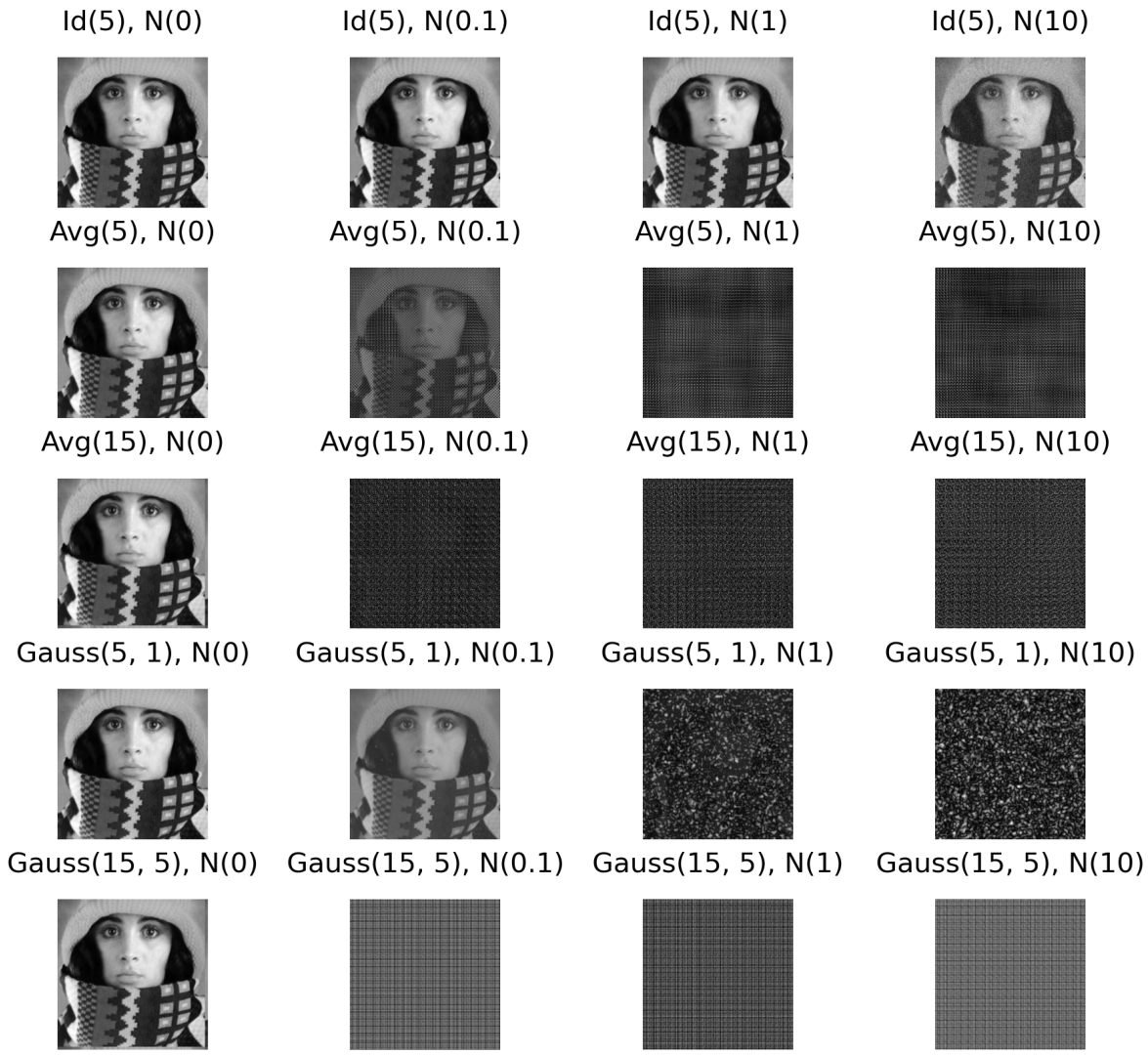

Figure 4: Degraded images from 3 after applying direct inverse filtering to them.

2.3

The Wiener filtering, under the assumption that the noise spectrum is constant, can be written as

$$\frac{1}{H'} = \frac{1}{H} \cdot \frac{|H|^2}{|H|^2 + K}$$

where H' is the best approximate solution to H when noise is present, given some appropriate constant K. The implementation of this can be seen in 7. Multiple values of K was attempted; the effect was that generally smaller values would not recover the original image very well, while higher values would recover them decently even with heavy noise present, but still with significant blurring. See 5 for the output of the Wiener filtering function as applied to the degraded images from task 2.1, using K = 0.05, which seems to be a good value to use here.

We see that the recovered images are much better than the direct inverse filtering when it comes to noisy images: all of the images regardless of how much noise or size of the kernel were able to be recovered to the point that they are at least fully recognizable. With the smaller kernels and low to moderate noise, the images are almost perfectly recovered.

However, it would seem that for the images with no noise present, the Wiener filtering is actually worse, as can be seen for the N = 15 kernels that are still blurry.

```
def ApplyWeinerFiltering(image, psf, K):
    image_ft = fft2(image)
    psf_ft = fft2(psf, image.shape)
    inv_filter = (1/psf_ft * (np.abs(psf_ft)**2 / (np.abs(psf_ft)
       **2 + K)))
    recovered_image = ifft2(np.multiply(image_ft, inv_filter))
    return np.roll(recovered_image, shift = (2, 2), axis = (0, 1))
```

Listing 3: Implementation of the Wiener filtering. Again, Numpy's roll function is there to properly align the images.

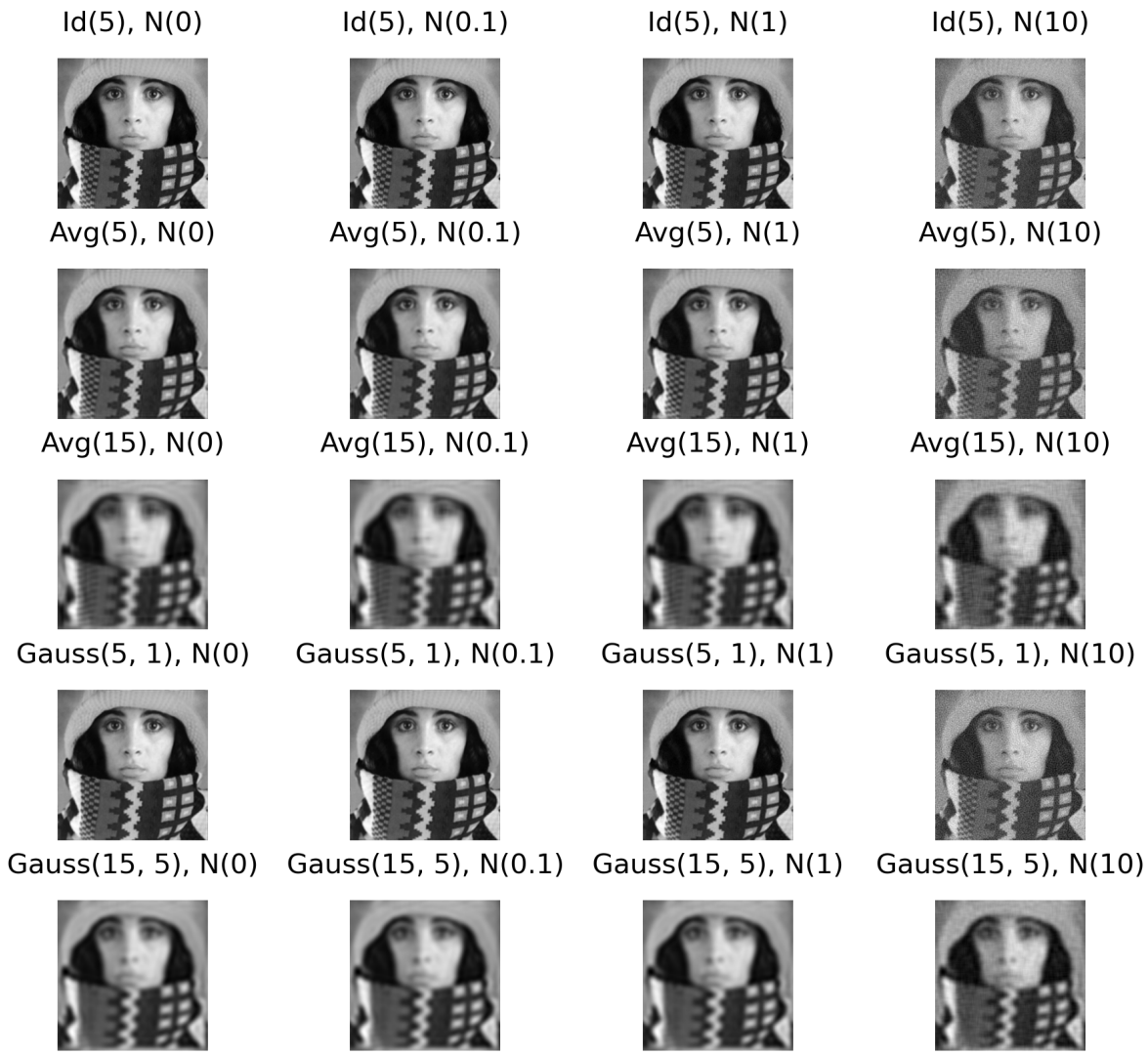

Figure 5: Results of applying the implemented Wiener filtering to the degraded images. Here, K = 0.05.

3. Transformations on images - Translation

3.1

A mathematical expression can be written for an image translated one pixel to the right as a function of the original image.

$$\tilde{I}(x, y) = I(x - 1, y) \tag{4}$$

3.2

The homogeneous matrix corresponding to this transformation is:

$$T = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

The transformation matrix can be used to produce $\tilde{I}(x, y)$ in the following way:

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + 1 \\ y \\ 1 \end{bmatrix} \tag{6}$$

Where $\tilde{x}$ and $\tilde{y}$ are the coordinates of $\tilde{I}$ and x and y are the coordinates for I. It can be seen that:

$$\tilde{x} = x + 1 \quad \Leftrightarrow \quad x = \tilde{x} - 1 \tag{7}$$

Which yields the equation in problem 3.1.

3.3

The same transformation can be done using a linear filter. In our case we have chosen to apply the filter using convolution. The filter kernel for translating an image 1 pixel to the right using convolution is:

$$\begin{bmatrix} \mathbf{0} & 1 \end{bmatrix} \tag{8}$$

Where the bold symbol denotes the center pixel which is the leftmost element in the kernel.

3.4

The following function creates a images with odd dimensions in x and y and centers a white square with NxN dimensions.

```
def centred_square(x, y, N):
    if x%2==0 or y%2==0:
        raise ValueError("The x and y dimensions of the image must
            be odd")
    output = np.zeros((x, y))
    x_center, y_center = x//2, y//2
    half_N = N//2
    if N%2==1:
        output[x_center-half_N:x_center+half_N+1, y_center-half_N:
           y_center+half_N+1] = 1
```

```
    else:
        output[x_center-half_N:x_center+half_N, y_center-half_N:
            y_center+half_N] = 1
    return output
```

The function was used to create a 9x9 image with a 3x3 centered white square. The resulting image can be seen in figure 6. The pixels are at the location where the blue grid lines intersect.

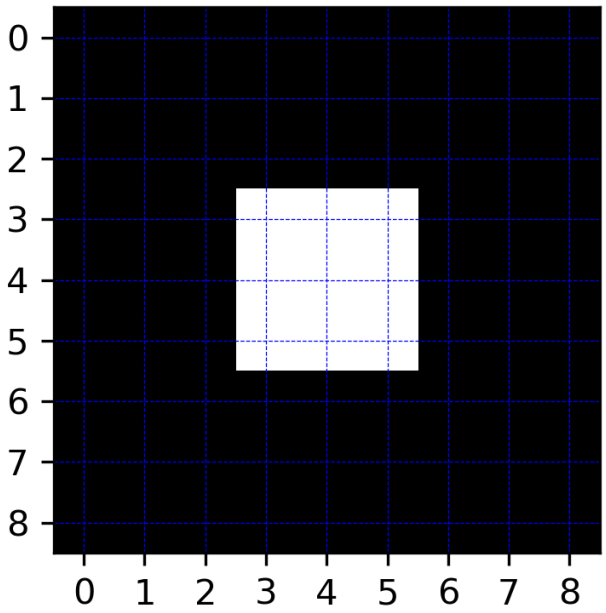

Figure 6: A 9x9 image is created with a centred 3x3 white square. The blue dotted lines are grid lines.

3.5

The following function utilizes a filter and convolution to do translation in x and y for a given image. The function was applied to do a translation of $t_x = 3$ and $t_y = 3$ on the image created in assignment 3.4. The resulting figure can be seen in figure 7.

```
def filter_translation(image, tx, ty, mode="full", boundary="
    fill"):
filter_x, filter_y = np.zeros(2*np.abs(tx)+1), np.zeros(2*np.
    abs(ty)+1)
filter_x[np.abs(tx)+tx] = 1
filter_y[np.abs(ty)+ty] = 1
filter = np.outer(filter_y, filter_x)
output = convolve2d(image, filter, mode, boundary)
return output
```

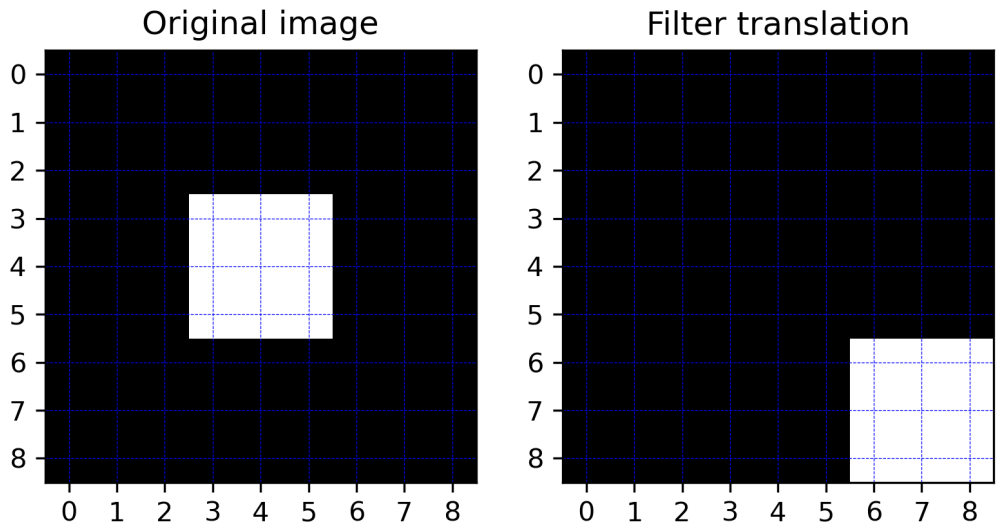

Figure 7: The filter translation function was used to translate the image in figure 6 by $t_x = 3$ and $t_y = 3$. The blue dotted lines are grid lines.

There are a number of different ways to handle the boundaries when doing translation. One thing that can be done is extending the image in the direction of translation, creating a larger image than the original. Another way is doing padding. One way to do padding is to do 0 padding. This makes it so that new pixels moved into the image have an intensity of 0. This also mean that information of the pixels moved outside of the image's dimensions are lost. Another way to do padding is by wrapping the image around and connecting the left and right side as well as the upper and lower border. This means that if a pixel is translated outside the image's dimension on the right side it will reenter on the left side of the image.

3.6

The following function uses a homogeneous matrix, backwards mapping and nearest neighbor interpolation to do translation. Due to the function using interpolation it can translate non-integers. The function was used to translate the original white square image by $t_x = 0.6$ and $t_y = 1.2$ which can be seen in figure 8. As nearest neighbor was utilized to do interpolation the effect is similar to rounding t_x and t_y if another interpolation method, like bilinear, was used the borders of the white square would blur into the background.

```
    def homogeneous_translation(image, tx, ty):
        output = np.zeros(image.shape)
        matrix = np.identity(3)
        matrix[0, 2] = tx
        matrix[1, 2] = ty
        matrix = np.linalg.inv(matrix)
        x = np.linspace(1, image.shape[1], image.shape[1])
        y = np.linspace(1, image.shape[0], image.shape[0])
        X, Y = np.meshgrid(x, y)
        for i in range(len(x)):
            for j in range(len(y)):
                loc = matrix@np.array([x[i], y[j], 1])
                x_nearest = int(round(loc[0]-1))
```

```
                y_nearest = int(round(loc[1]-1))
                if x_nearest >= image.shape[0] or y_nearest >=
                   image.shape[1]:
                    output[j, i] = 0
                else:
                    output[j, i] = image[y_nearest,x_nearest]
        return output
```

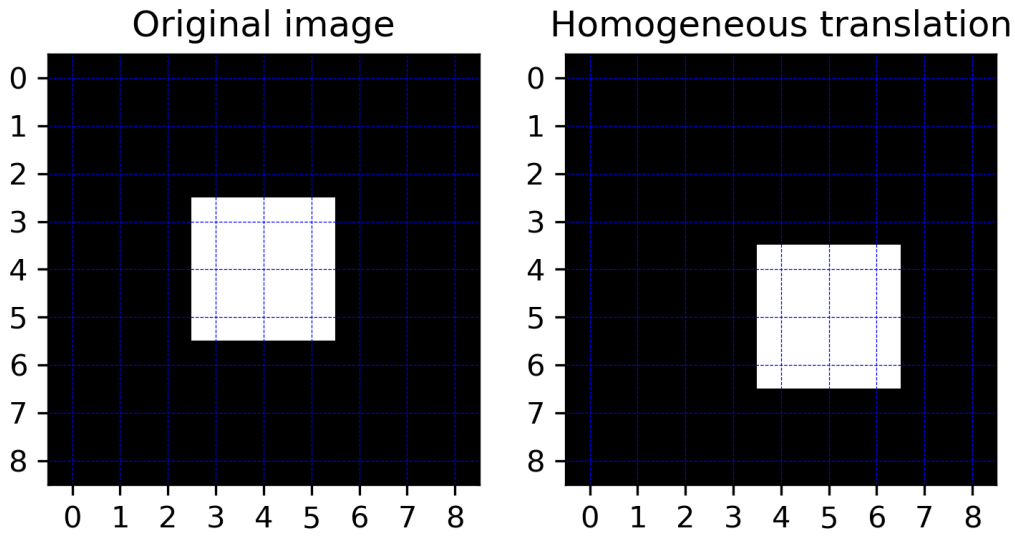

Figure 8: The homogeneous translation function was used to translate the image in figure 6 by $t_x = 0.6$ and $t_y = 1.2$. The blue dotted lines are grid lines.

3.7

The following function utilizes the formula for the power spectrum of a translated image to do translation using the Fourier transform. The function was applied to the original white square image with $t_x = 3$ and $t_y = 3$ and the result can be seen in figure 9.

```
    def fourier_translation(image, tx, ty):
    fft_image = fft2(image)
    u = np.fft.fftfreq(fft_image.shape[1])
    v = np.fft.fftfreq(fft_image.shape[0])
    u, v = np.meshgrid(u, v)
    rotate = np.exp(-2j*np.pi*(u*tx+v*ty))
    return np.real(ifft2(fft_image*rotate))
```

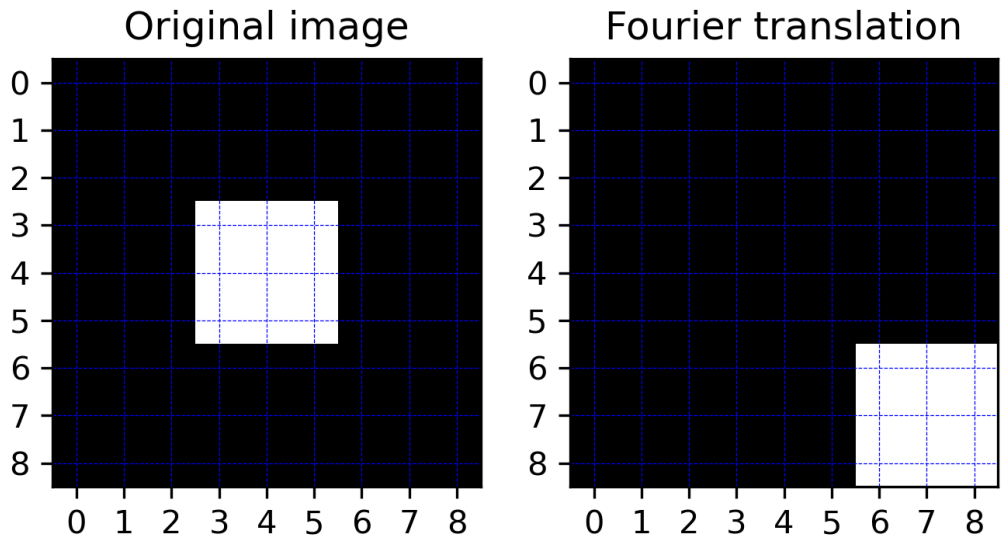

Figure 9: The Fourier translation function was used to translate the image in figure 6 by $t_x = 3$ and $t_y = 3$. The blue dotted lines are grid lines.

When doing the same translation using the filter method and Fourier transform method with $t_x = 3$ and $t_y = 3$ the two methods yields the same result. However dependent on the boundary conditions chosen for the filter method the result can diverge if the objects in the image is translated outside the image's dimensions. The result from the Fourier method corresponds to a filter translation with the wrap boundary condition. If 0 padding was chosen for the filter image the results of the two methods would not be the same.

3.8

The Fourier method was used to translate a 33x33 image with a 7x7 centred square by $t_x = 5.6$ and $t_y = 7.2$. The resulting image can be seen in figure 10.

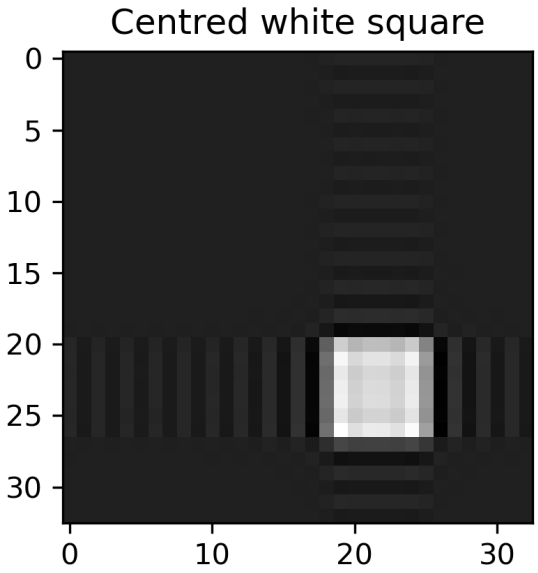

Figure 10: The Fourier translation function was applied to a 33x33 with a centred 7x7 square. The image was translated $t_x = 5.6$ and $t_y = 7.2$.

The result is quite different from doing integer translation. There are horizontal and vertical damped oscillations originating in the new position of the square. The square seems to be in roughly the correct position but it is not completely white anymore. The Fourier transformation method can therefore not perform non-integer translations perfectly.

A non-integer translation was also applied to "cameraman.tif" using the Fourier transformation method. The image was translated by $t_x = 15.3$ and $t_y = 30.7$ and the result can be seen in figure 11.

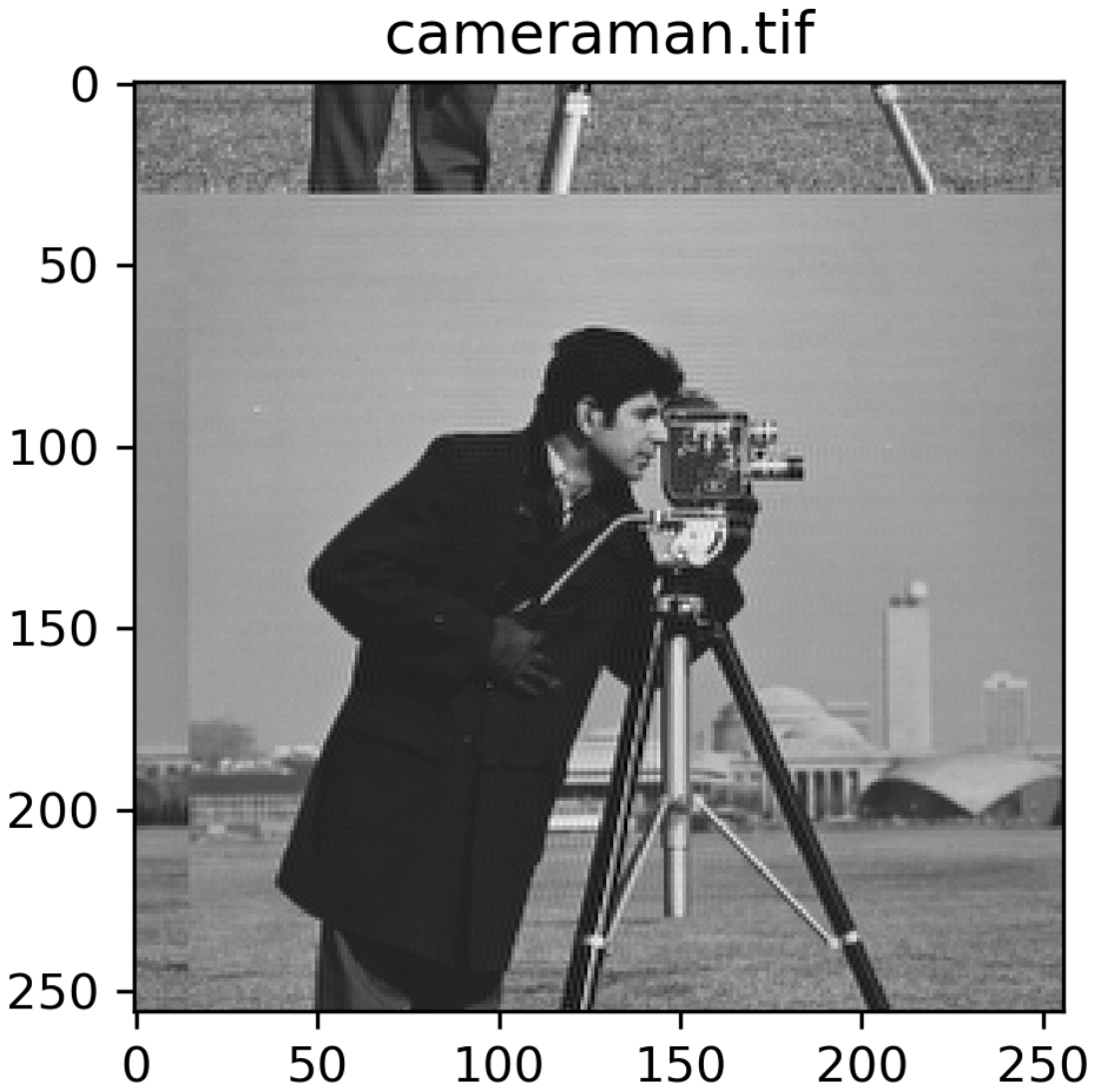

Figure 11: The Fourier translation function was applied to the "cameraman.tif" image. The image was translated $t_x = 15.3$ and $t_y = 30.7$.

On the image it can be seen that there are some small artifacts from the translation resulting in some vertical and horizontal blur. However this is not as apparent as in the white square image.