UNIVERSITY OF COPENHAGEN

COMPUTER SCIENCE

SIGNAL AND IMAGE PROCESSING

Assignment 4

Author:
Casper BRESDAHL

Teachers:
Kim PEDERSEN
Sune DARKNER

March 8, 2021

1 Exercise 1

1.1

When performing correlation and convolution we look at each pixel in the image at turn. The 'x' filter in this exercise looks at the pixel to the right and weights it by $1/2$, it looks at the pixel to the left and weights it $-1/2$, but the center pixel is not apparent in the formula, and is thus weighted zero. The same goes for the 'y' filter of the exercise. Thus we have:

'x' filter for convolution:

1/2	0	-1/2

'x' filter for correlation:

-1/2	0	1/2

'y' filter for convolution:

1/2
0
-1/2

'y' filter for correlation:

-1/2
0
1/2

For all the filters the center pixel is the zero in the kernel. There is no difference between correlation and convolution if the kernel is symmetric. Given a Gaussian kernel is symmetric and a mean filter is constant (and thus also symmetric) there is no difference between convolution and correlation with these filters.

1.2

Linearly separating the Sobel y derivative filter gives us:

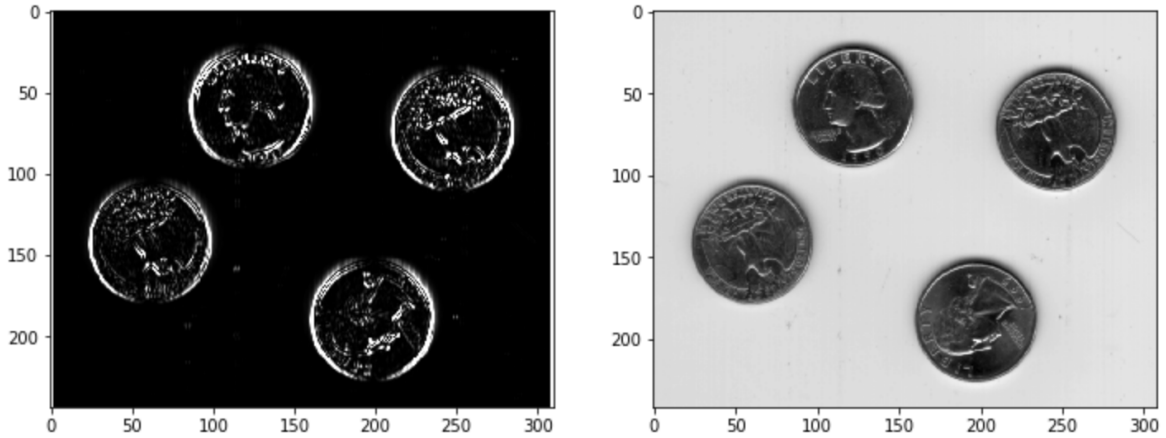
$$f * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = f * \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

Linearly separating the Prewitt y derivative filter gives us:

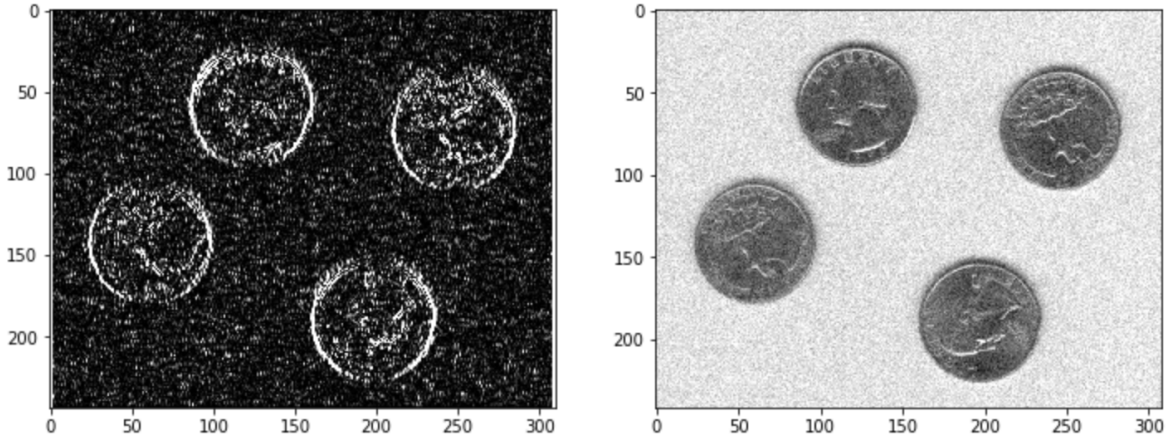
$$f * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = f * \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

We see that the separated y filter is almost identical to the one from the previous exercise, that is, we get almost the same result when convolving with the y filter alone. But for the Sobel and Prewitt filters we also convolve with an x filter afterwards, this means we involve more pixels in the convolution, and 'smooth' the noise more. We can look at the x filter as a 'smoothing' filter, as we are smoothing the image afterwards by adding the neighbouring pixel values to the center pixel. Because we involve more pixels in the convolution, the noisy pixels gets weighted less in total.

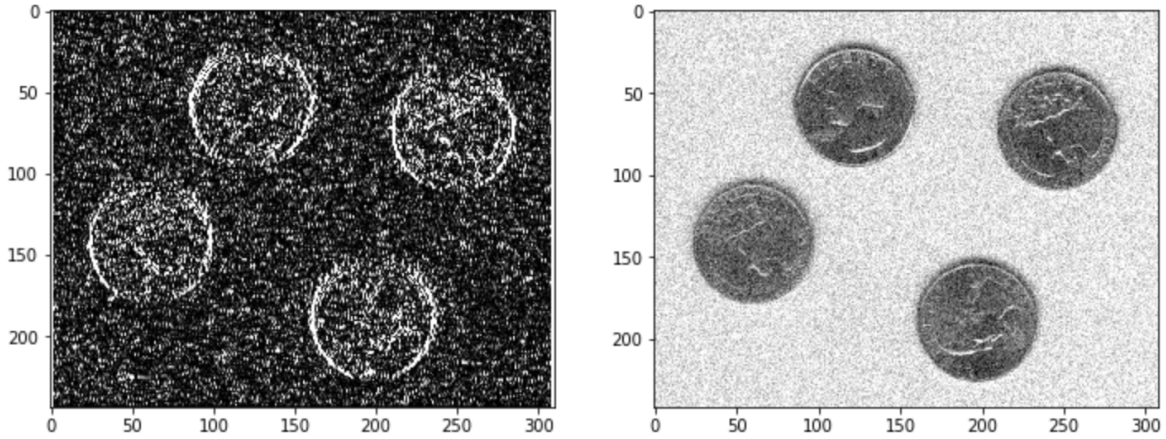
1.3



(a) Variance 0

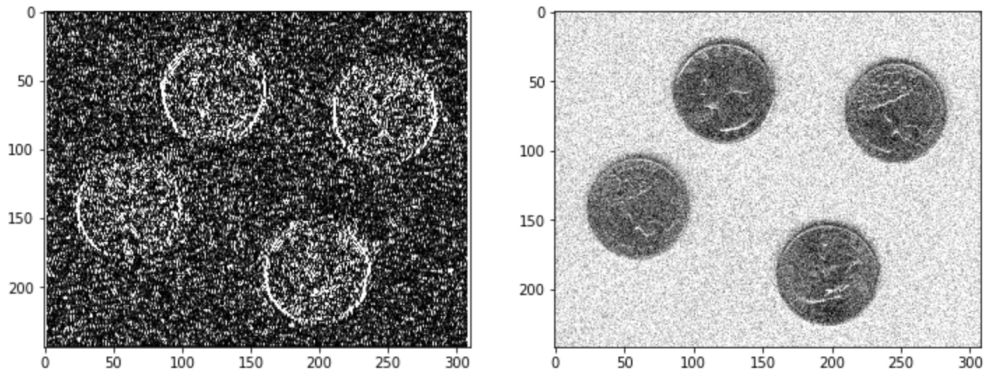


(b) Variance 0.005

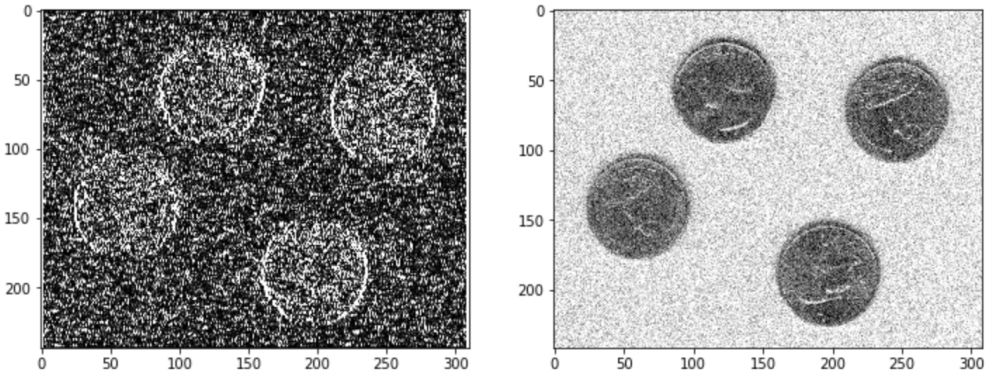


(c) Variance 0.01

Figure 1: Gradients produced by filtering with Sobel.

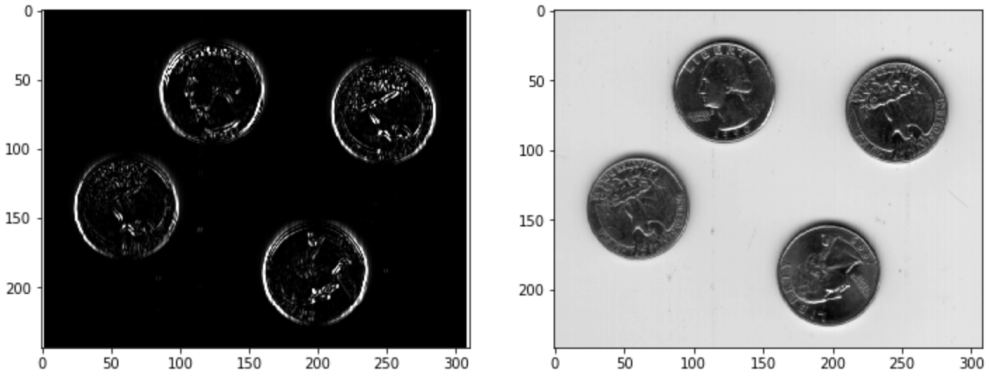


(a) Variance 0.015

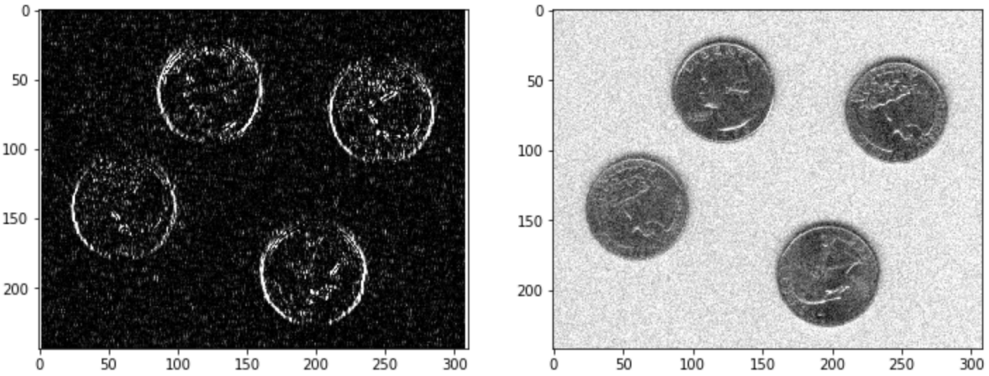


(b) Variance 0.02

Figure 2: Gradients produced by filtering with Sobel.

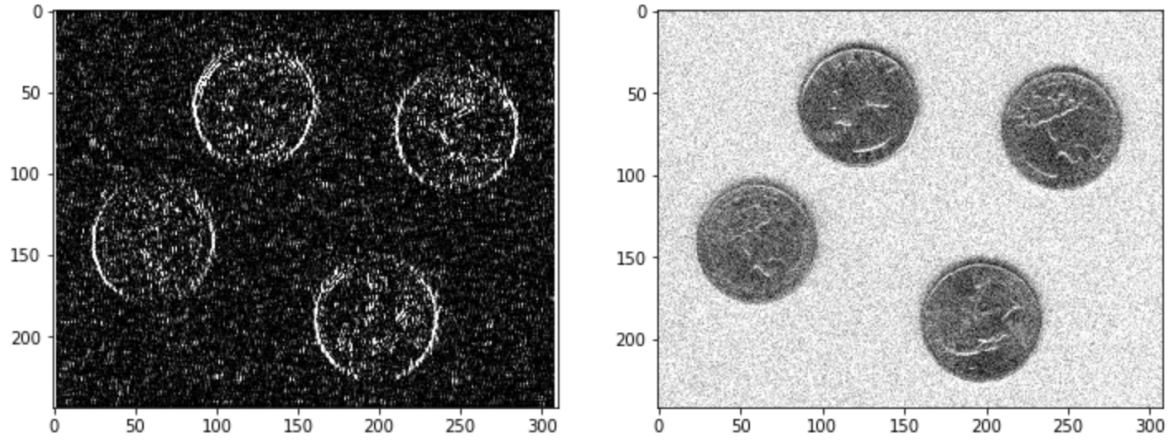


(a) Variance 0

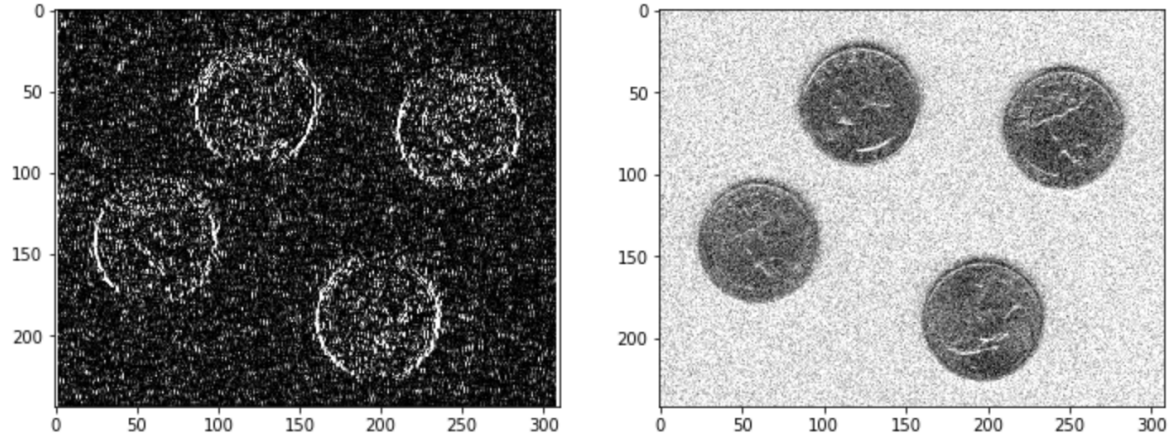


(b) Variance 0.005

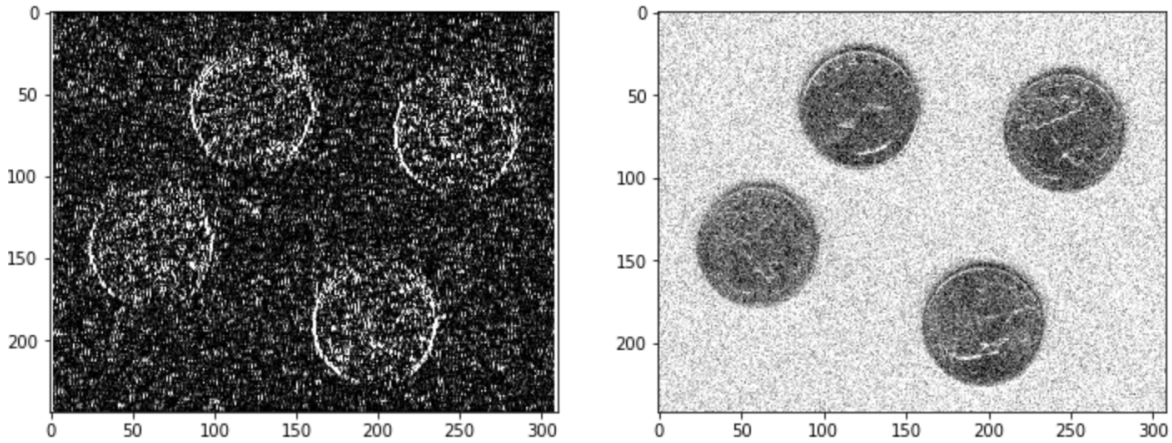
Figure 3: Gradients produced by filtering with Prewitt.



(a) Variance 0.01



(b) Variance 0.015



(c) Variance 0.02

Figure 4: Gradients produced by filtering with Prewitt.

In Figure 1, Figure 2, Figure 3 and Figure 4 we see filtering with the Sobel and Prewitt filter for different variances. We see that the Prewitt filter seems to less effected by noise than the Sobel filter. Comparing the images with 0.2 variance it becomes somewhat hard to see the actual gradients with the Sobel filter whereas it is still rather clear in the Prewitt filter. However, the gradient is much more clear in the low variance images for the Sobel filter, where a lot more detail is also caught.

The code used for this exercise can be seen in Figure 5.

```
def gradientImg(im, xk, yk):
    varz = np.arange(0, 0.025, 0.005)
    for var in varz:
        print(var)
        noise = rn(im, mode='gaussian', var=var)
        dx = ss.convolve2d(noise, xk)
        dy = ss.convolve2d(noise, xk)
        fig, ax = plt.subplots(1,2, figsize=((12,16)))
        print(ax.shape)
        ax[0].imshow(dx**2 + dy**2, cmap='gray',vmin=0,vmax=1)
        ax[1].imshow(noise, cmap='gray')
        plt.show()
```

Figure 5: Code for exercise 1.

2 Exercise 2

2.1

To prove $\tilde{I}_1(x,y) = \tilde{I}_2(x,y) = \frac{a+b}{2}$ we first look into the CDF and the inverse CDF for the images. For a constant image, the CDF would be 0 until it reaches the constant and then it spikes to 1. If we compute $CDF(I(x,y))$ of a constant image we will get 1 for all the pixels because all the pixels have the same intensity. The inverse CDF gives us the value corresponding to the probability. For a constant image, if we take $CDF(I(x,y))$ we get the probability 1, and if we then take $CDF^{-1}(1)$ we get the intensity of the constant image.
For this exercise we have C_1 and C_2. Taking $C_1(I_1(x,y))$ gives us 1 as I_1 is constant. Taking $C_1^{-1}(C_1(I_1(x,y)))$ thus simply gives us the pixel intensity of $I_1(x,y) = a$ back. But taking $C_2^{-1}(C_1(I_1(x,y)))$ gives us the constant value of I_2 back, as C_2 spikes at the constant value of I_2 which is b.
We thus have:

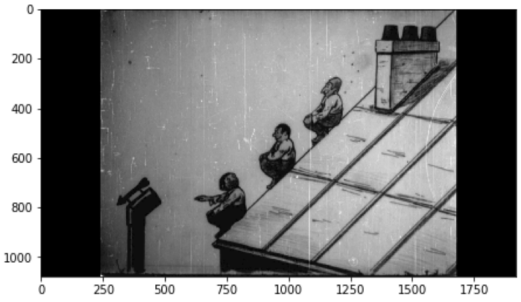
$$\begin{aligned}\tilde{I}_1 &= \frac{1}{2}\left(C_1^{-1}(C_1(I_1)) + C_2^{-1}(C_1(I_1))\right)\\ &= \frac{1}{2}\left(C_1^{-1}(1) + C_2^{-1}(1)\right)\\ &= \frac{1}{2}(a+b)\end{aligned}$$

And:

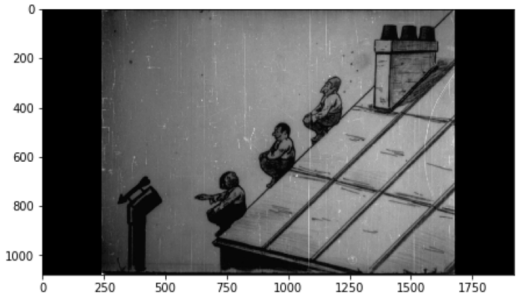
$$\begin{aligned}\tilde{I}_2 &= \frac{1}{2}\left(C_1^{-1}(C_2(I_2)) + C_2^{-1}(C_2(I_2))\right)\\ &= \frac{1}{2}\left(C_1^{-1}(1) + C_2^{-1}(1)\right)\\ &= \frac{1}{2}(a+b)\end{aligned}$$

And we have thus proved $\tilde{I}_1(x,y) = \tilde{I}_2(x,y) = \frac{a+b}{2}$.
As $\frac{a+b}{2}$ gives us the average value, the resulting CDF will spike at the average, and thus the cumulative histogram of the midway specification is equal to the average of the cumulative histograms.

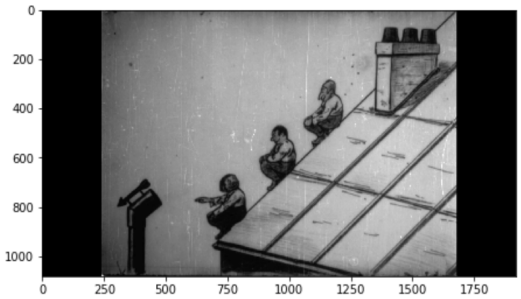
2.2



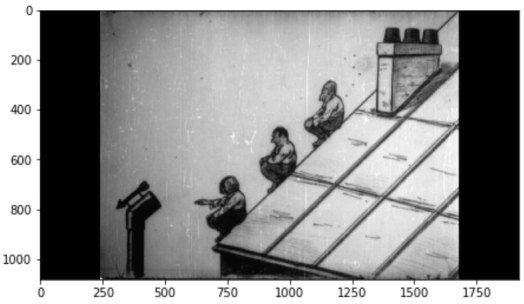
(a) Midway specification of 'movie_flicker1' image.



(b) Original 'movie_flicker1'.

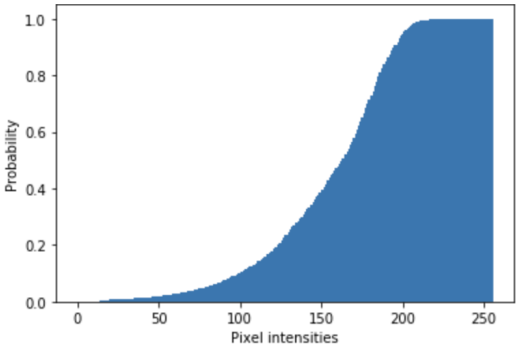


(c) Midway specification of 'movie_flicker2' image.

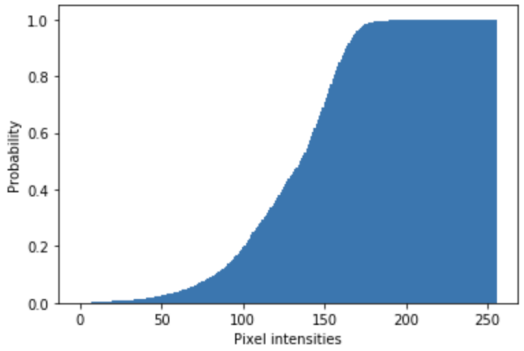


(d) Original 'movie_flicker2'.

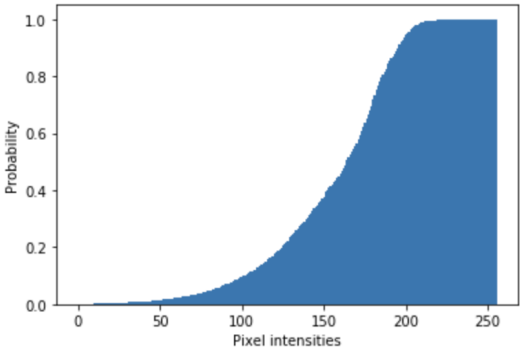
Figure 6: Midway specification and original images.



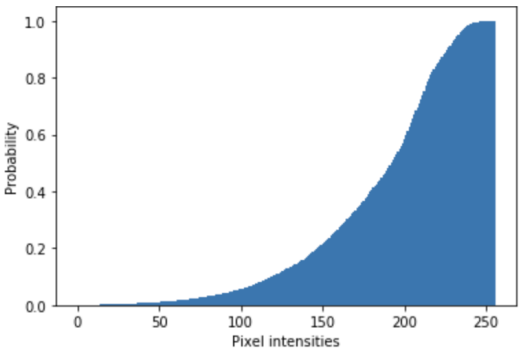
(a) CDF of Midway specification of 'movie_flicker1' image.



(b) CDF of Original 'movie_flicker1'.



(c) CDF of Midway specification of 'movie_flicker2' image.



(d) Original 'movie_flicker2'.

Figure 7: CDF of Midway specification and original images.

In Figure 6 we see the midway specifications of the two images *'movie_flicker1'* and *'movie_flicker2'* together with the original images. In Figure 7 we see their CDFs. To generalize the midway specification function ϕ we would define it as:

$$\phi(x) = \frac{1}{N}\left(C_1^{-1}(x) + C_2^{-1}(x) + ... + C_N^{-1}(x)\right)$$

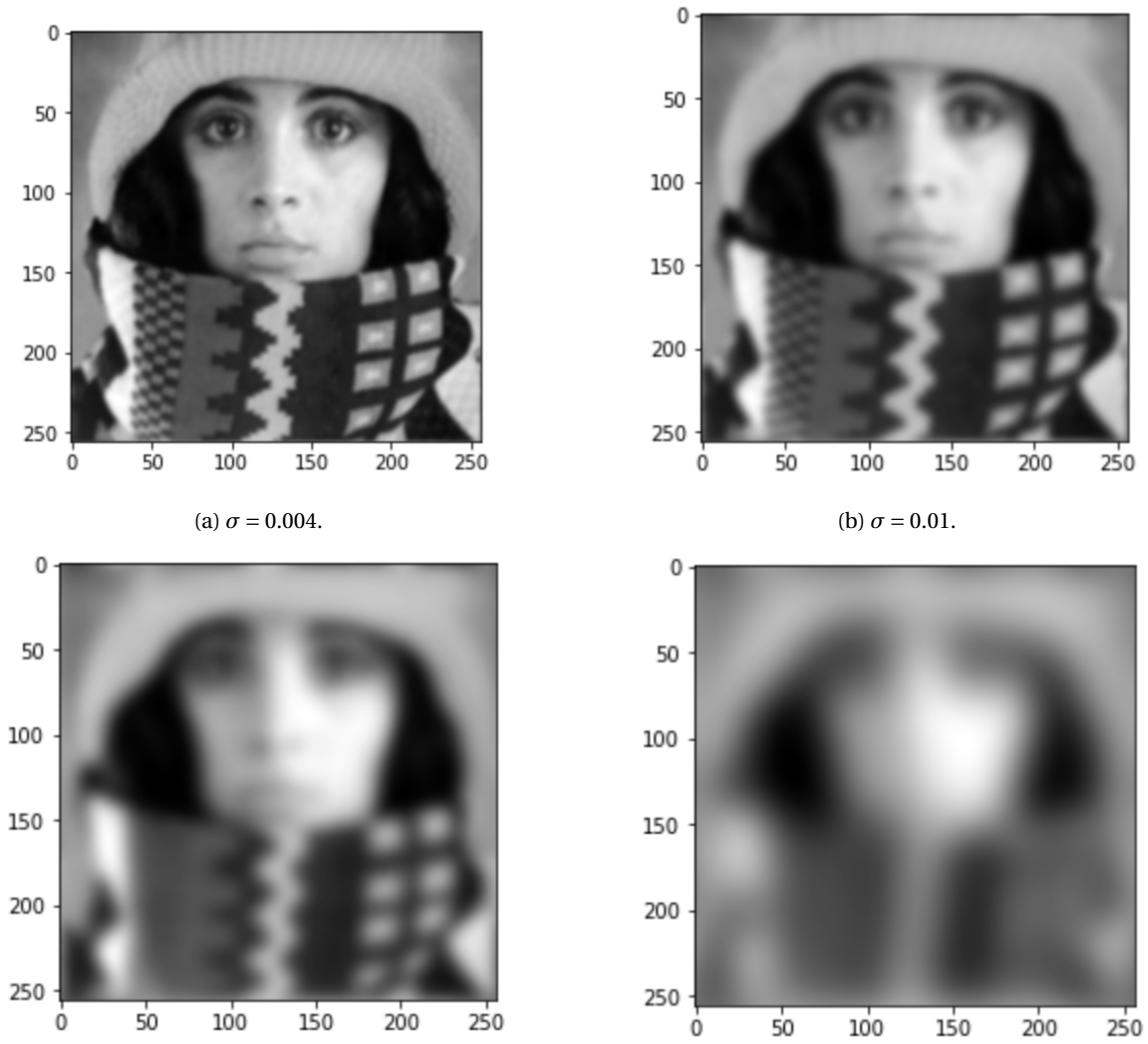
Where N is the number of images. We would then get the n'th midway specification as $\tilde{I}_n = \phi(C_n(I_n))$. The code for this exercise can be seen in Figure 8. The 'cdf' function returns a list of probabilities (one for each pixel intensity) and the corresponding pixel intensities. The utility functions not shown have been reused from last week's assignment.

```
def midway2(im1, im2):
    s1,i1 = cdf(im1)
    s2,i2 = cdf(im2)

    mdws1 = np.zeros_like(im1)
    mdws2 = np.zeros_like(im2)

    for y in range(im1.shape[0]):
        for x in range(im1.shape[1]):
            pi1 = im1[y,x]
            pi2 = im2[y,x]
            inv11 = inverse((s1,i1), s1[pi1])
            inv21 = inverse((s2,i2), s1[pi1])
            inv12 = inverse((s1,i1), s2[pi2])
            inv22 = inverse((s2,i2), s2[pi2])

            mdws1[y,x] = (inv11 + inv21) / 2
            mdws2[y,x] = (inv12 + inv22) / 2
    return mdws1, mdws2
```

Figure 8: Code for exercise 2.

3 Exercise 3

3.1



(a) $\sigma = 0.004$.

(b) $\sigma = 0.01$.

(c) $\sigma = 0.02$.

(d) $\sigma = 0.05$.

Figure 9: Gaussian kernel constructed in frequency space multiplied by Fourier transform of *'trui'*.

From the convolution theorem we know that convolution in the spatial domain is equal to multiplication in frequency domain. That is:

$$\mathscr{F} f * g(u) = \mathscr{F} f(u) \cdot \mathscr{F} g(u) = F(u) \cdot G(u)$$

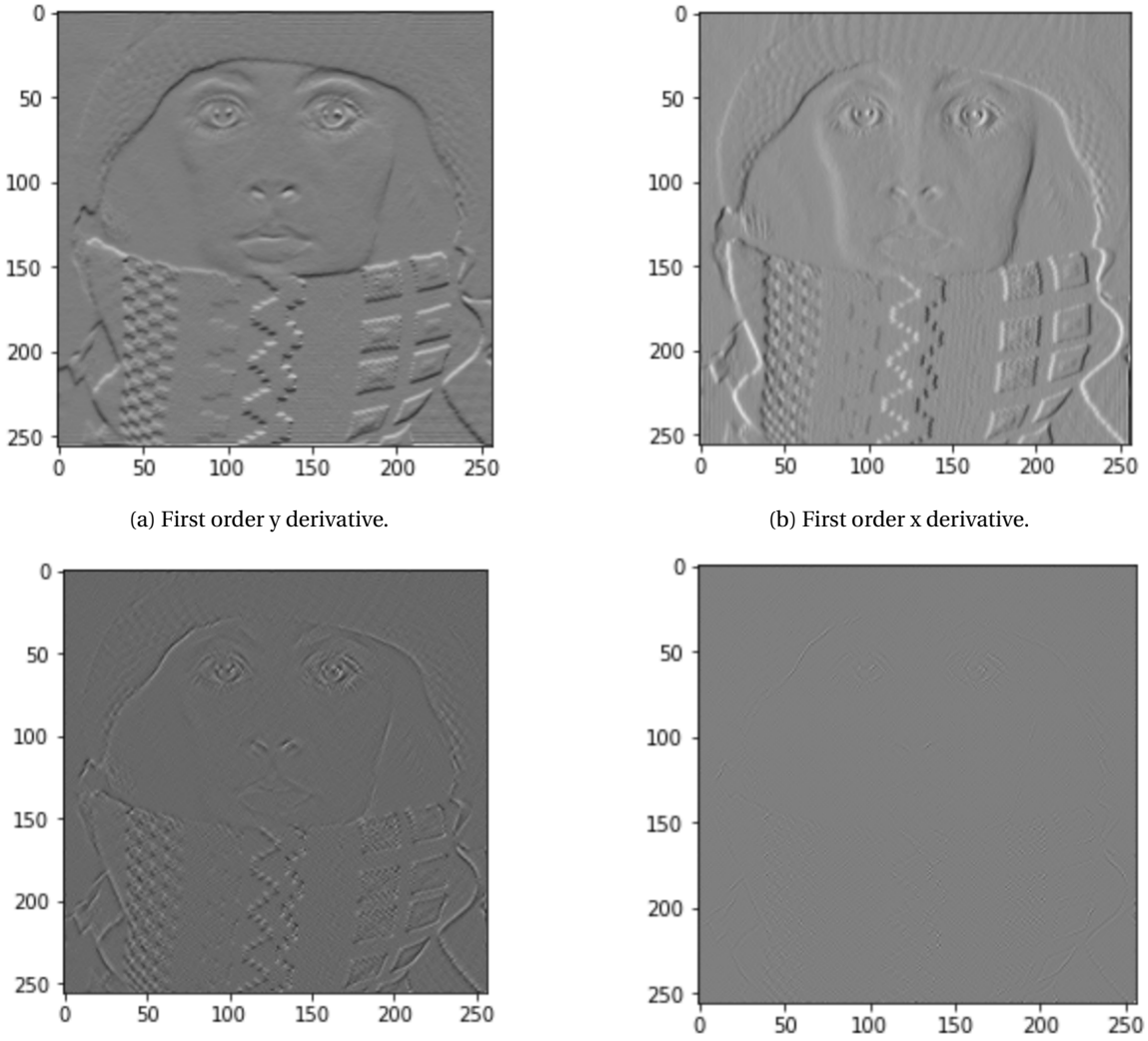
Thus, if we want to perform convolution between two functions, we can instead find both functions Fourier transform, multiply these together, and then take the inverse Fourier transform of the result.
In Figure 9 we see the result of constructing a Gaussian kernel in frequency space and multiply it with the Fourier transform of *'trui.png'* and then take the inverse Fourier transform of the result. The code used for this exercise can be seen in Figure 10. The kernel constructed by *'fftgausK'* is defined based on the Fourier transform of a Gaussian kernel given in the lecture slides.

[7]
```
def fftgausK(sigma):
    n = 0.5/sigma
    x, y = np.mgrid[-n:n+1, -n:n+1]
    H = np.exp(-2*np.pi**2 * sigma**2 * (x**2 + y**2))
    return H
```

[14]
```
def scale_fft(img,k):
    sz = (img.shape[0] - k.shape[0], img.shape[1] - k.shape[1])
    k = np.pad(k, (((sz[0]+1)//2, sz[0]//2), ((sz[1]+1)//2,sz[1]//2)), mode='constant')
    k = ifftshift(k)
    fft = fft2(img)
    fft_filtered = fft * k
    f = ifft2(fft_filtered)
    return np.real(f)
```

Figure 10: Code for exercise 3.1.

3.2



(a) First order y derivative.

(b) First order x derivative.

(c) First order x and y derivative.

(d) Second order x and y derivative.

Figure 11: Partial derivatives of *'trui'* image.

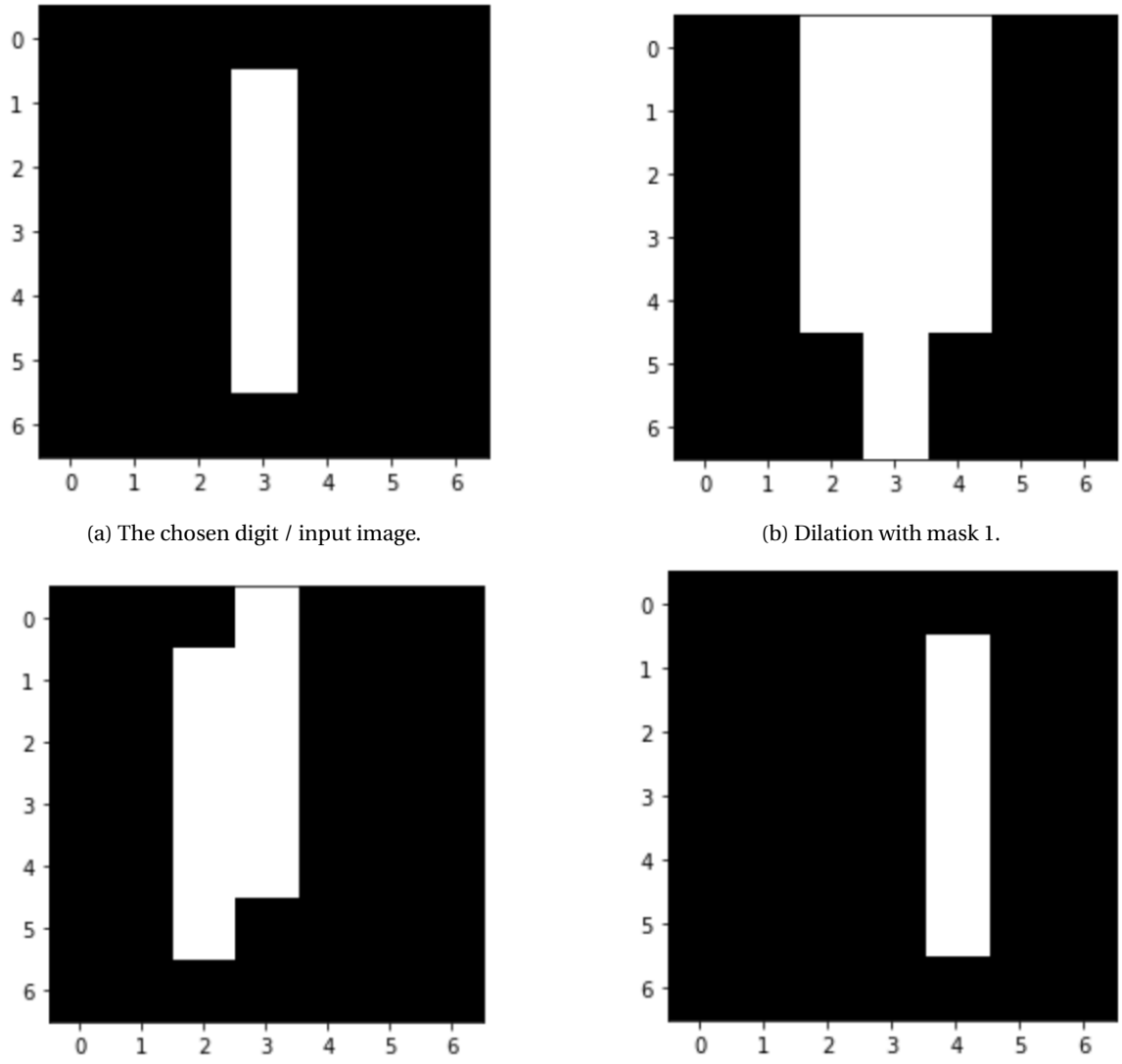
From the lecture slides we have the derivative theorem which tells us $\frac{\partial^m}{\partial x^m}\frac{\partial^n}{\partial y^n}f(x,y)$ in spatial domain is equivalent to $(iu)^m(iv)^bF(u,v)$ in the frequency domain. That is, we can construct a kernel based on $(iu)^m(iv)^b$, multiply by $F(u,v)$ and then take the inverse Fourier transform to get our partial derivatives. The results can be seen in Figure 11 and the code can be seen in Figure 12.

```python
def deriv(im, a, b):
    n1 = (im.shape[0]+1)//2
    n2 = (im.shape[1]+1)//2
    u,v = np.mgrid[-n1:n1,-n2:n2]
    k = (1j*u)**a * (1j*v)**b
    k = fftshift(k)

    fft = fft2(im)
    filtered = fft * k
    return ifft2(filtered)
```

Figure 12: Code for exercise 3.2.

4 Exercise 4

4.1



(a) The chosen digit / input image.

(b) Dilation with mask 1.

(c) Dilation with mask 2.

(d) Dilation with mask 3.

Figure 13: Results of dilation with masks from the exercise.

i.: The structuring element does not need to be symmetric. As can be seen in Figure 13b the resulting image after dilation adds the 'T' shape to the input image. The centre pixel is defined as the middle one in the 3 by 3 grid, i.e. the white pixel in the second row.

ii.: The structuring element does not need to have an odd number of pixels in both directions. In Figure 13c we see the diagonal pattern added to the input image. The centre pixel is the lower right black pixel.

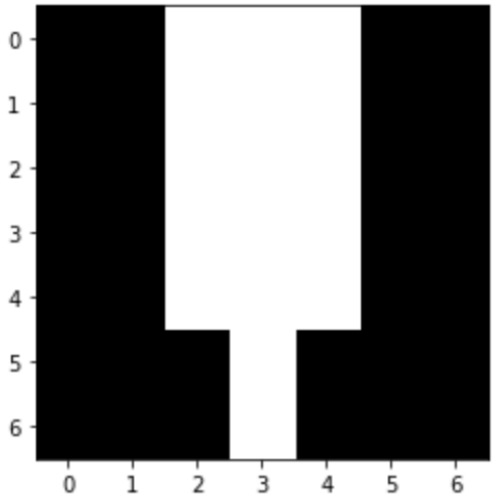
iii.: The structuring element does not need to have the middle pixel set to true. As seen in Figure 13d the output simply gets shifted. The centre pixel is the middle black pixel.

The code for this exercise can be seen in Figure 14.

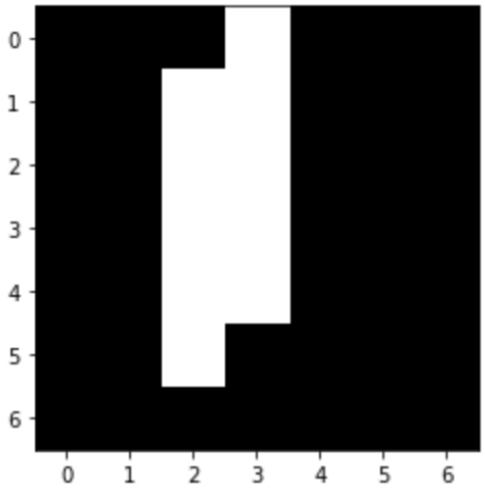
```
A4 = np.array([0,0,0,0,0,0,0,
               0,0,0,1,0,0,0,
               0,0,0,1,0,0,0,
               0,0,0,1,0,0,0,
               0,0,0,1,0,0,0,
               0,0,0,1,0,0,0,
               0,0,0,0,0,0,0]).reshape((7,7))
plt.imshow(A4,cmap='gray')
plt.show()

m1 = np.array([[1,1,1],
               [0,1,0],
               [0,1,0]])
plt.imshow(m1,cmap='gray')
plt.show()

m2 = np.array([[0,1],
               [1,0]])
plt.imshow(m2,cmap='gray')
plt.show()

m3 = np.array([0,0,1]).reshape((1,3))
plt.imshow(m3,cmap='gray')
plt.show()
```

Figure 14: Code for exercise 4.1.

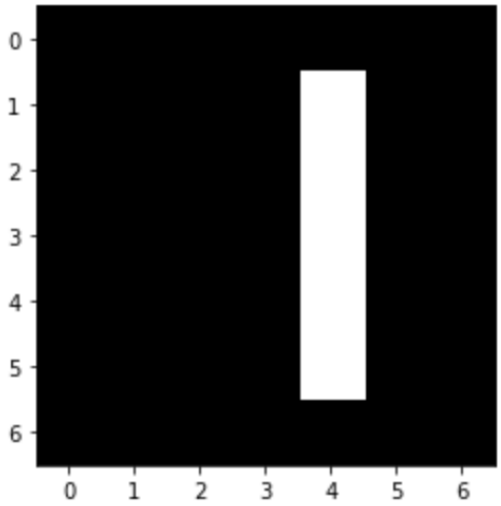
4.2



(a) Mask 1.



(b) Mask 2.



(c) Mask 3.

Figure 15: Results of dilation of masks with input image.

When performing the dilation with the input image on the masks we get the exact same results as before due to the commutative property of dilation. This indeed means we can interchange the input image and mask and get the same results. This can be useful if either the mask or the input image contains *many* more 'true' pixels than the other, then interchanging the mask and the input image such that we only need to perform dilation on a few 'true' pixels might speed things up.
The results of this exercise can be seen in Figure 15 and the code for this exercise can be seen in Figure 16.

```
plt.imshow(bd(np.pad(m2, (2,3)), A4), cmap='gray')
plt.show()
plt.imshow(bd(np.pad(m1, 2), A4), cmap='gray')
plt.show()
plt.imshow(bd(np.pad(m3, ((3,3),(2,2))), A4), cmap='gray')
plt.show()
```

Figure 16: Code for exercise 4.2.