UNIVERSITY OF COPENHAGEN

COMPUTER SCIENCE

SIGNAL AND IMAGE PROCESSING

Assignment 8

Author:
Casper BRESDAHL

Teachers:
Kim PEDERSEN

April 12, 2021

1 Exercise 1

1.1

When using homogenous coordinates we can dot the transformation matrices together to form a single transformation rather than having to apply several transformation in turn. The only slight downside is the conversion between Cartesian and homogenous coordinates, which however is very easy to perform.

1.2

A rotation has one degree of freedom namely θ, scaling has 2 degrees of freedom, the x scale and the y scale, and translation has 2 degrees of freedom, the x translation and the y translation. In total we thus have 5 degrees of freedom. The points needed to determine the translation is 1, as we can simply move the the point that is to be aligned. The rotation needs one more point to be determined as we can rotate around the point we translated until the second point is in place. And scaling takes another point to determine as the scaling could be negative and thus be mirrored. In total we thus need $N = 3$ to determine the alignment.

2 Exercise 2

2.1

If we denote the pixels in the image I as x' and y' and we denote the pixels in $\tilde{I}$ as x and y, we can map the pixels in $\tilde{I}$ to the pixels in I by taking each pixel pair, x y, augment with a 1, and then multiply with the inverted transformation matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos(2\pi-\theta) & -sin(2\pi-\theta) & 0 \\ sin(2\pi-\theta) & cos(2\pi-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \right)^{-1} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The resulting augmented pixel pair, x' y', are then the corresponding x and y coordinates in the image I. Running through all pixels in $\tilde{I}$, performing this mapping and then copying the pixel value from I to $\tilde{I}$ would result in the correct transformation.

We can also compute the inverted transformation matrix. Multiplying the three transformation matrices results in the following matrix:

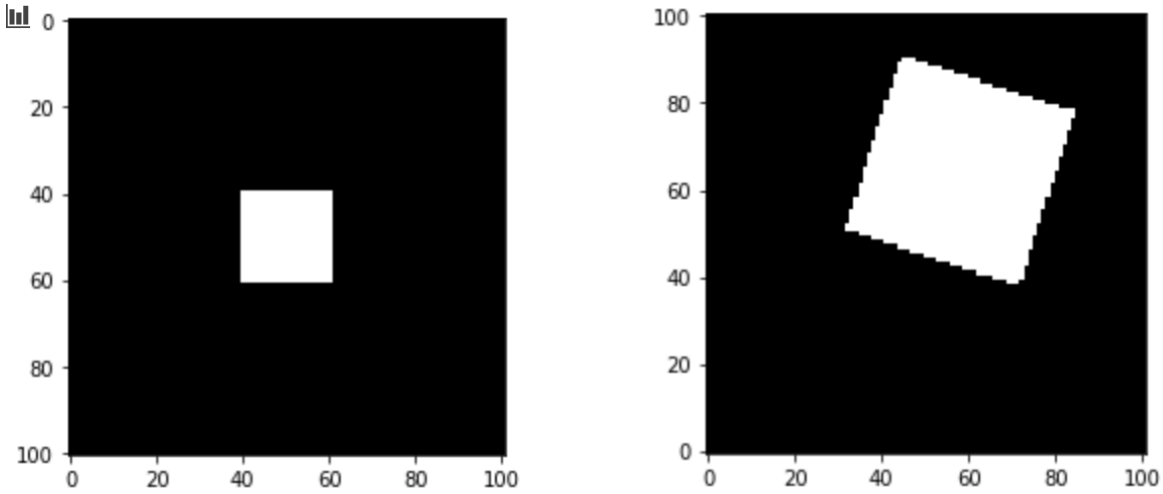
$$\begin{bmatrix} s \cdot cos(2\pi-\theta) & s \cdot -sin(2\pi-\theta) & t_x \\ s \cdot sin(2\pi-\theta) & s \cdot cos(2\pi-\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

To invert this matrix we need to perform 4 steps. In the first step we compute the matrix of minors. That is, for each entry in turn, we ignore the current row and current column, and then compute the determinant of the remaining elements which then replaces the entry we are at in the matrix. In step two we then change sign of all the elements so we get a checkerboard pattern of positive and negative values. This is the matrix of cofactors. In step three we transpose the cofactors matrix. And in the last step we multiply all elements

with $\frac{1}{det} = \frac{1}{(s \cdot cos(2\pi-\theta))^2 + (s \cdot -sin(2\pi-\theta) - (s \cdot sin(2\pi-\theta)))}$. We then get the matrix:

$$\frac{1}{det} \cdot \begin{bmatrix} s \cdot cos(2\pi-\theta) & -(s \cdot -sin(2\pi-\theta) & (s \cdot -sin(2\pi-\theta) \cdot t_y) - (s \cdot cos(2\pi-\theta) \cdot t_x) \\ -(s \cdot sin(2\pi-\theta) & s \cdot cos(2\pi-\theta) & (s \cdot cos(2\pi-\theta) \cdot t_y) - (s \cdot sin(2\pi-\theta) \cdot t_x) \\ 0 & 0 & (s \cdot cos(2\pi-\theta))^2 - (s \cdot sin(2\pi-\theta) \cdot s \cdot -sin(2\pi-\theta)) \end{bmatrix}$$

2.2



(a) Before transformation

(b) After transformation

Figure 1: Before and after transformation of white square.

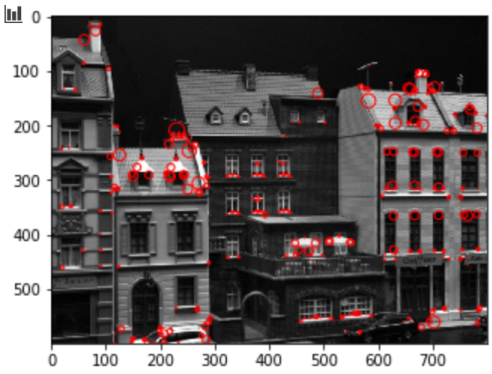
In Figure 1 we see the white square before and after the transformation. In Figure 2 we see the code used for this exercise. We first define the transformation matrices in homogenous coordinates and then dot them all together, and then we take the inverse of this matrix as he final transformation matrix. We then loop through all pixels in the output image and map them to the input image through the inverted transformation matrix. To make the dotting of the transformations work, we use 3x3 matrices, and thus the input coordinates needs to be augmented with a third coordinate equal to 1. The output coordinate is also augmented, and to convert back to Cartesian coordinates we thus divide the x and y coordinates with the augmented output coordinate, however, this augmented output coordinate is always 1, so we simply just get the x and y coordinate. To accommodate the nearest neighbour interpolation, we simply round the coordinates. After we have performed the transformation, we plot the new square where we have changed the direction of the y-axis to be of correct orientation.

```python
def transform_image(im):
    cos = np.cos(2*np.pi - np.pi/10)
    sin = np.sin(2*np.pi - np.pi/10)
    s = 2
    Y,X = im.shape

    T_t = np.array([[1,0,10.4],
                    [0,1,15.7],
                    [0,0,1]])
    T_c = np.array([[1,0,X//2],
                    [0,1,Y//2],
                    [0,0,1]])
    T_c_inv = np.array([[1,0,-X//2],
                        [0,1,-Y//2],
                        [0,0,1]])
    R = np.array([[cos,-sin,0],
                  [sin, cos,0],
                  [0,0,1]])
    S = np.array([[s,0,0],
                  [0,s,0],
                  [0,0,1]])

    t = T_t@T_c@R@S@T_c_inv

    old_im = im
    new_im = np.zeros_like(im)
    for y in range(Y):
        for x in range(X):
            nx, ny, _ = np.dot(np.linalg.inv(t),[x,y,1])
            nx = int(np.round(nx))
            ny = int(np.round(ny))
            new_im[y,x] = old_im[ny,nx]
    return new_im
```

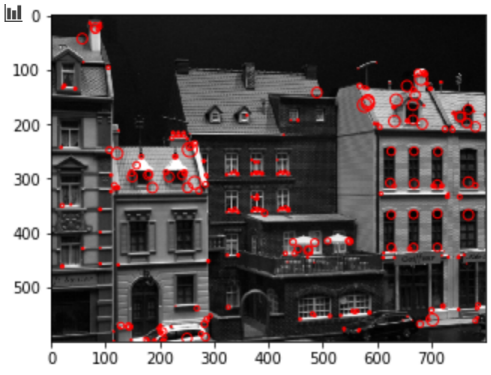
Figure 2: Code used for this exercise.

3 Exercise 3

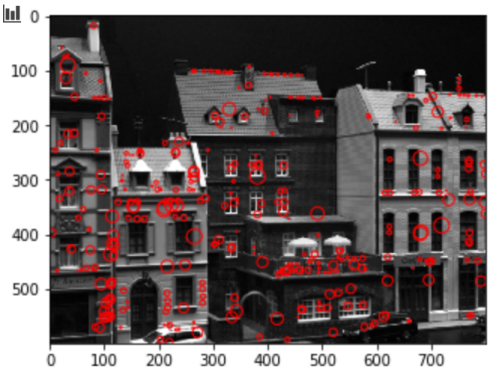
3.1



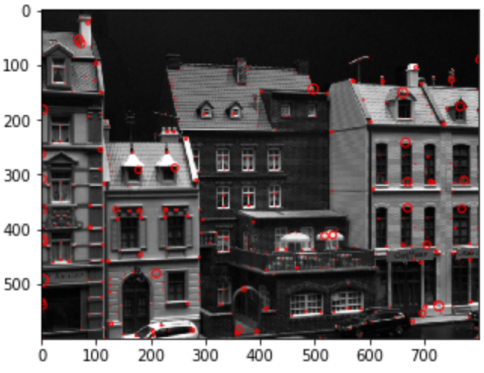
(a) $k = 1$, $\alpha = 0.05$.



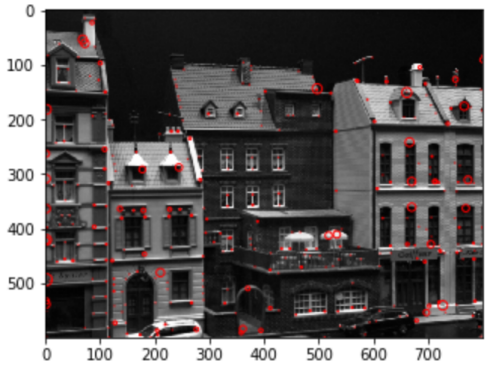
(b) $k = 1$, $\alpha = 0.15$.



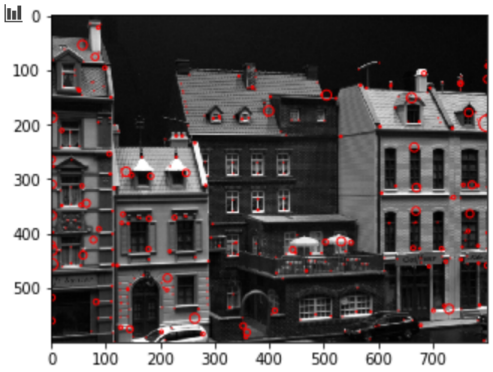
(c) $k = 1$, $\alpha = 0.95$.



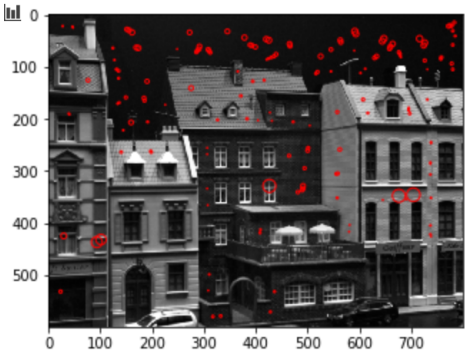
(d) $k = 2$, $\alpha = 0.15$.



(e) $k = 4$, $\alpha = 0.05$.



(f) $k = 4$, $\alpha = 0.15$.



(g) $k = 4$, $\alpha = 0.95$.

Figure 3: Multi -scale Harris corner detection with various k and α values.

In Figure 3 we see the results of the multi-scale Harris corner detection for various values of k and α. The range of scales is 30 evenly spaced values between 2^0 and 2^5 as advised in the assignment text. In Figure 4 we see the code used for this exercise. We see that if we increase the α value too much we begin to detect edges rather than corners. We also see if increase the combination of k and α too much (i.e. the last image) we start to get unusable results. We find the best result to be with k around 2 and α around 0.15.

```
scale_levels = 30
scales = np.logspace(0,5,scale_levels,base=2)
k = 4
alpha = 0.05
modelhouse = modelhouse / np.max(modelhouse)
res = []

print(len(scales))

for scale in scales:
    A = construct_A(modelhouse,scale,k).T
    det = np.linalg.det(A)
    T = alpha*np.trace(A.T)**2
    R = scale**4 * (det.T - T)
    res.append(R)

extrema = []
copy = np.array(res)
z,y,x = copy.shape
zero = np.zeros((y,x))
print(copy.shape)
stack = np.pad(copy, 1, mode='constant', constant_values=(0))
print(stack.shape)
for i in range(1,z+1):
    print(f'{i}',end='\r')
    for j in range(1,y+1):
        for k in range(1,x+1):
            vs = np.delete(stack[i-1:i+2,j-1:j+2,k-1:k+2], [13])
            if(stack[i,j,k] > np.max(vs)):
                extrema.append([i,j,k, stack[i,j,k]])
```

```
S = np.array(sorted(extrema, key=abs_sort)[-350:])
print(S.shape)
fig, ax = plt.subplots()
ax.imshow(modelhouse,cmap='gray')
zipped = zip(S[:,2],S[:,1],S[:,0],S[:,3])
for x,y,r,v in zipped:
    c = plt.Circle((x,y),r*0.5,color='red', fill=False)
    ax.add_patch(c)
plt.show()
```

Figure 4: Code used for this exercise.

4 Exercise 4

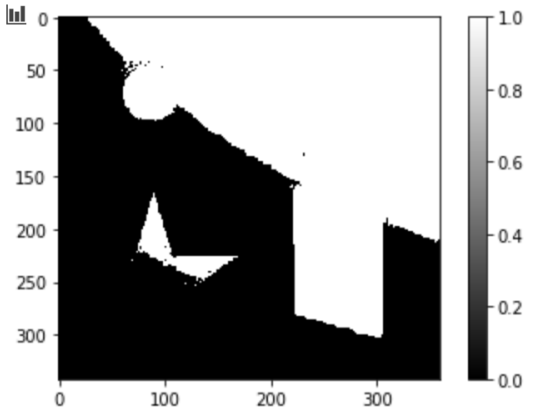
4.1

When we are working with images with a background which gradually changes intensity, i.e. when some of the background is light, and some of the background is dark, it can be hard for intensity based segmentation approaches to segment correctly. However, an edge based approach would perform well. An example would be Figure 5 (first image taken from segmentation lecture slides).

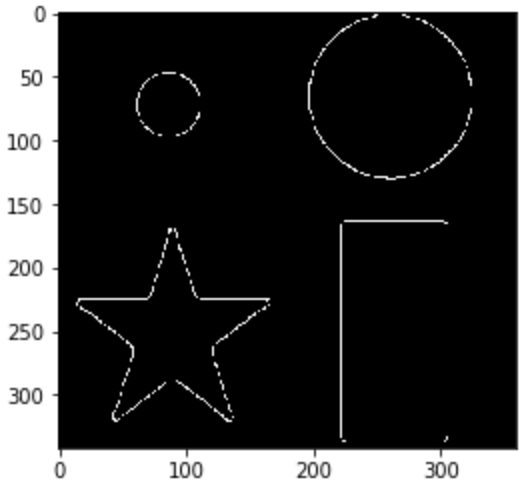
Intensity based approaches are also problematic when we are segmenting blurred images, as the edges becomes gradually 'sharp' and thus, a small change in the threshold might result in a big difference in the size of the segmented object. Also what previously was right angles now becomes more soft. An example of this can be seen in the images from the segmentation lecture slides in Figure 6.

A more broad issue is the consistency of intensity based approaches, as the lighting of the scene might brighten or darken the image (this can both be uniformly and non-uniformly). That is, if we develop an automatic pipeline of image processing where we process several images of the same scene, but with different lighting, we can not expect the same amount of quality in the segmentation. This could be a big issue in a self driving car which should be able to drive during both day and night.

(a) Image with gradually changing background.

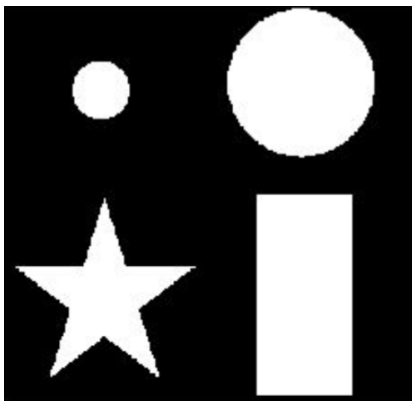


(b) Intensity based segmentation approach. Threshold at 75.

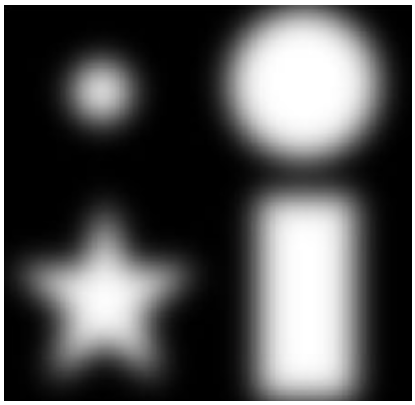


(c) Canny edge detection. $\sigma = 3$, low threshold = 0 and high threshold = 10.

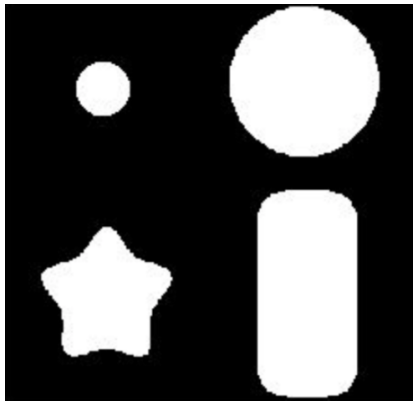
Figure 5: Example 1 of problems with intensity based segmentation.



(a) Sharp example image.



(b) Blurred example image.



(c) Example image after intensity based segmentation.

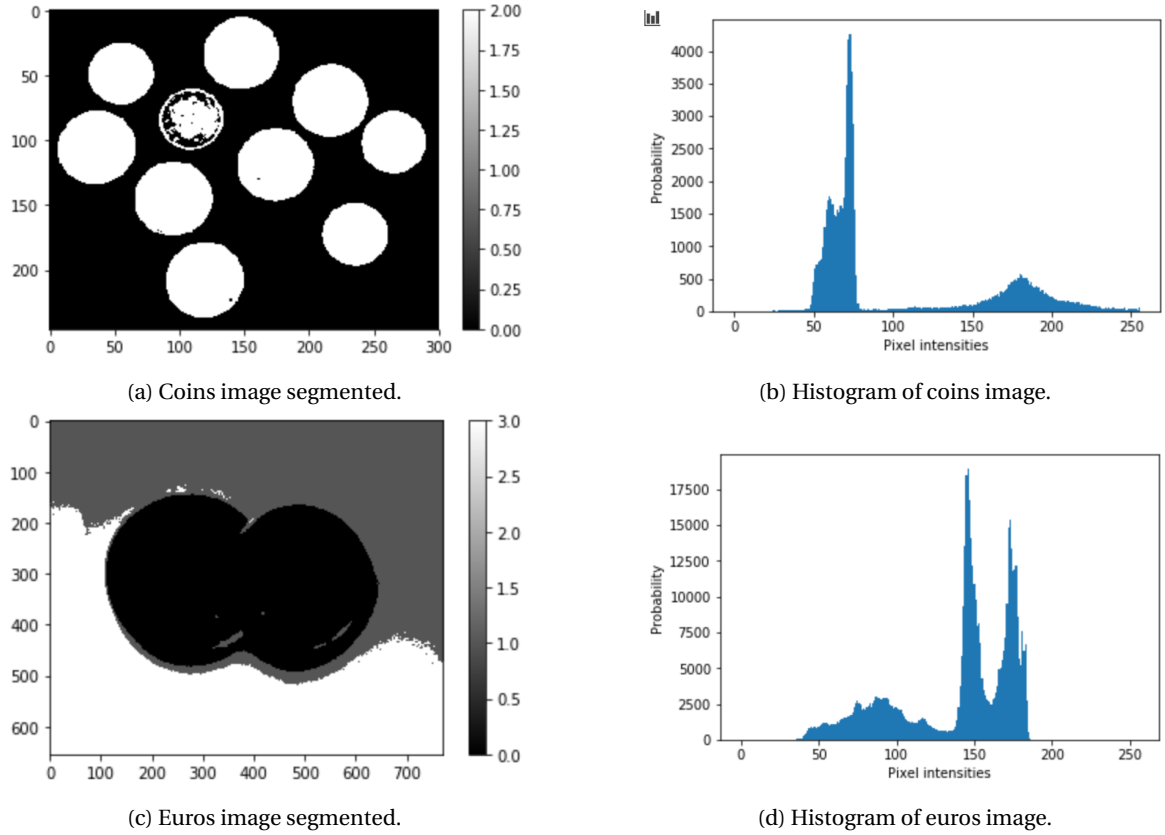
Figure 6: Example 2 of problems with intensity based segmentation.

4.2

Edge based segmentation is likely to perform better in the cases of abrupt discontinuities, i.e. when we have clear object boundaries, shadows and specularities or in general when the scene has not been evenly lit. A more concrete example would be the segmentation of Figure 5 where an intensity based approach most likely would not segment the star and the circle correctly at the same time.

Intensity based approaches would however perform better with noisy images as it is unlikely that clear edges can be detected as the derivatives probably will 'drown' in the noise. Intensity based approaches would not achieve perfect segmentation, but the general shape would probably be present.

4.3



(a) Coins image segmented.

(b) Histogram of coins image.

(c) Euros image segmented.

(d) Histogram of euros image.

Figure 7: Results of automated histogram segmentation.

In Figure 7 we see the results of automatically segmenting the coins and euros image. In Figure 8 we see the code used for this exercise. To find the peaks in the histogram, we go through all intensity values and for each intensity we check the past 20 intensities, and the future 20 intensities and check if the amount of pixels with this intensity is the maximum over these intensities. This way we only get one peak despite the number of pixels close to each other in intensity fluctuates bit. We also check whether at least 0.05% of the total amount of pixels have this intensity. Both the width we search with and the percentage of total pixels we want minimum are adjustable parameters, but I found these values worked well for our two test images. Next we want to determine the thresholds. For this we want one threshold less than there are peaks, and we want these thresholds to be roughly in the middle between two peaks. For this reason, we in turn find the mean intensity between the current peak we are looking at and the next. We search ± 15 intensities to find the intensity with least pixels to find the most natural break. When we no longer have a 'next' peak, we 'group' all pixel intensities above the last threshold we found. As we see in Figure 7 the histogram of the coins image has two peaks which the function finds and performs a good segmentation of the images. Only a few pixels in two of the coins has been misclassified and the very dark coin is as expected not segmented as well as the rest of the coins. In the euros histogram we see three peaks, one dark quite flat, and two high intensity peaks. The result is a segmentation of the two coins as one grouping, the lower part as one grouping and the upper part as the last grouping. The result is as expected as the upper part of the image is quite a bit darker then the lower part.

With this approach we can quite easily find a small to medium amount of peaks and segment these. We also get results which are as expected from a human perspective when we are looking at peaks in the histograms, that is, the function does not report a peak every time the histogram has a local maxima, but only when these local maxima are sufficiently spaced from each other. We can see this in both histograms in Figure 7 where from a human perspective we would only consider the left most part of the histogram of the coin image as one peak and not two. And the same for the right most part of the euros image (where we from a human perspective would consider 3 peaks, and not 4). However, the function has its limitations. If the peaks becomes too uniform, the function would have a hard time separating the real peaks and the 'false'-local-maxima-peaks. But also if we only have one peak the function would have issues. One peak could appear if we have a histogram with the shape of a very peaky normal distribution. Then we could imagine the left part of the image being black, gradually becoming grey, and then the right part of the image gradually becoming white. From a human perspective, this would need three groupings, but the function would segment the whole image as one grouping.

As we use the histogram to select the thresholds, and the only difference between a histogram of a low resolution image and a high resolution image is the numbers along the y-axis, the quality of the image has a low influence on the results of the segmentation. If we upped the resolution of the images we would get more total pixels, and perhaps more pixels where there already are a lot of pixels as the distribution of the histogram would remain unchanged. This means it perhaps would be easier to identify peaks, but as we already require a peak to consist of some percentage of the total pixels to be considered a peak, it would not have much influence.

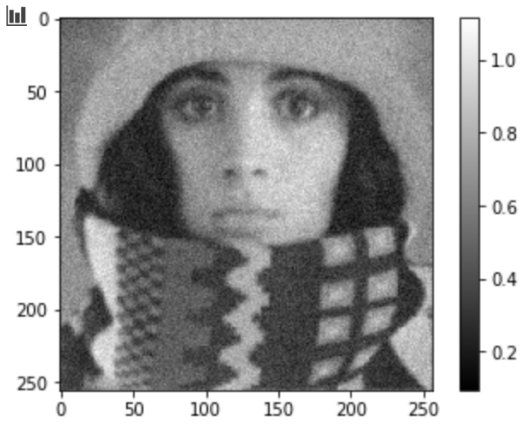
```python
def find_peaks(im, h, thresh=0.005, width=20):
    counts, vals = h
    Y, X = im.shape
    pixels = Y*X
    peaks = []

    for i in vals:
        min_val = np.maximum(0,i-width)
        max_val = np.minimum(255,i+width)
        val = counts[i]
        val_range = counts[min_val:max_val]
        if(np.all(val >= val_range) and (val / pixels) > thresh):
            peaks.append(i)
    return peaks
```

```python
def find_thresh(im, peaks, h):
    counts, vals = h
    num = len(peaks)
    prev = -1
    colors = np.linspace(0,num,num)
    copy = np.zeros_like(im)
    Y,X = im.shape

    for i in range(num-1):
        mean = (peaks[i] + peaks[i+1]) // 2
        min_val = np.maximum(0,mean-15)
        max_val = np.minimum(255,mean+15)
        val_range = counts[min_val:max_val]
        thresh = vals[(np.argmin(val_range) - 15) + mean]

        for y in range(Y):
            for x in range(X):
                if(prev < im[y,x] <= thresh):
                    copy[y,x] = colors[i]
        prev = thresh
    for y in range(Y):
        for x in range(X):
            if(prev < im[y,x] <= 255):
                copy[y,x] = colors[-1]
    return copy
```

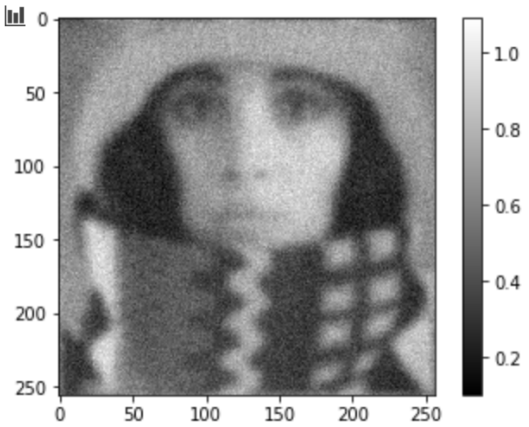
Figure 8: Code used for this exercise.

5 Exercise 5

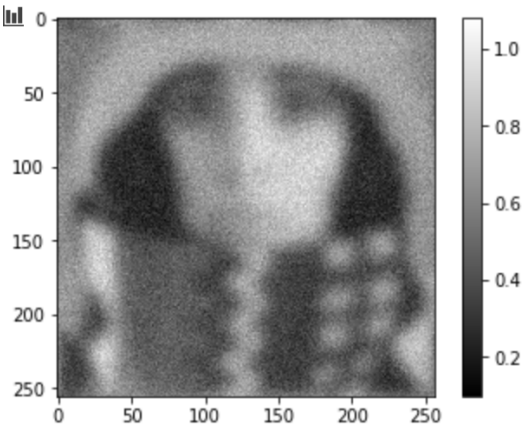
5.1



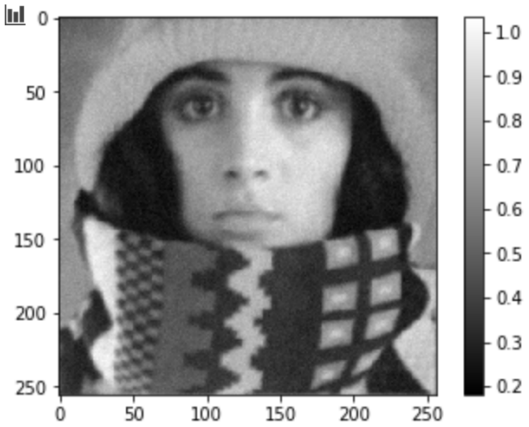
(a) Gaussian kernel with $\sigma = 1$ and noise with standard deviation of 0.05.



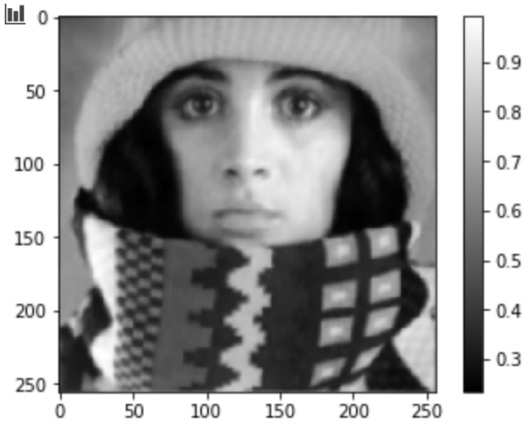
(b) Gaussian kernel with $\sigma = 3$ and noise with standard deviation of 0.05.



(c) Gaussian kernel with $\sigma = 5$ and noise with standard deviation of 0.05.



(d) Gaussian kernel with $\sigma = 1$ and noise with standard deviation of 0.02.



(e) Gaussian kernel with $\sigma = 1$ and noise with standard deviation of 0.001.

Figure 9: Results of linear shift invariant degradation model with different noise and blurring factors.

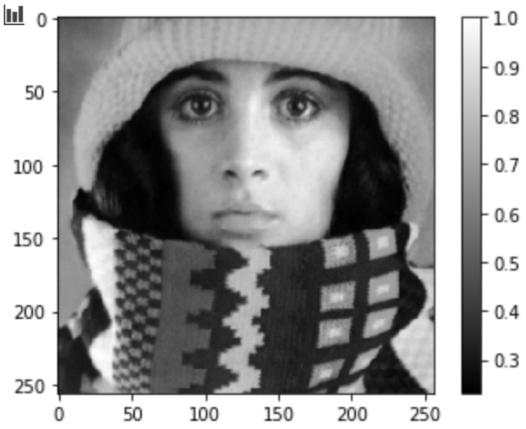
In Figure 9 we see the results of the linear shift invariant degradation model. In the first three images the σ values of the kernel is changing while the noise is kept constant, and in the last two images the σ is kept constant while we turn down the noise levels. The realization of the noise is created by drawing as many

samples with *np.random.normal* as there are pixels in the image with a mean of 0. We control the strength of the noise by tuning the standard deviation. To make the noise visual we normalize the trui image so all values are between 0 and 1. We see the noise is quite significant when the standard deviation is around 0.05, it is visible around 0.02 and almost gone at around 0.001. The code used for this exercise can be seen in Figure 10.

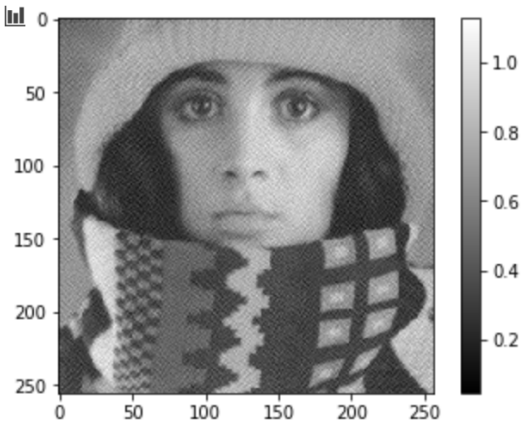
```
def white_noise(im,scale=0.05):
    Y,X = im.shape
    samples = np.random.normal(loc=0, scale=scale, size=Y*X)
    noise_source = samples.reshape((Y,X))
    return noise_source

M↓

def LSI(im,k,ns):
    return scale_fft(im,k) + ns
```

Figure 10: Code used for this exercise. *scale_fft* is a utility function which convolves an image and a kernel in frequency space.

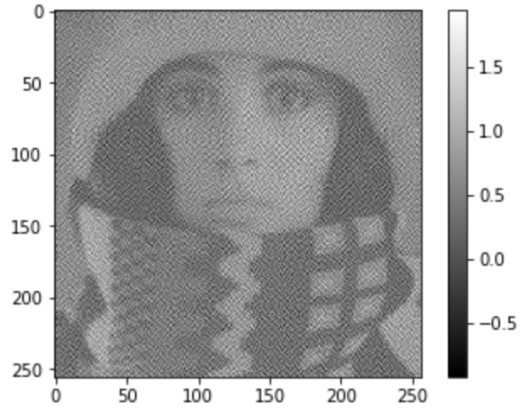
5.2



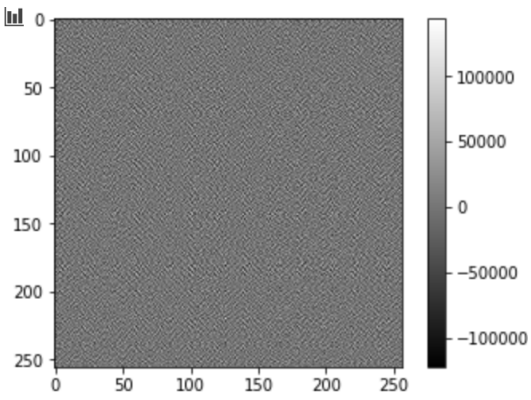
(a) Gaussian kernel with $\sigma = 3$ and no noise.



(b) Gaussian kernel with $\sigma = 3$ and noise with standard deviation of 0.000000002.



(c) Gaussian kernel with $\sigma = 3$ and noise with standard deviation of 0.00000001.



(d) Gaussian kernel with $\sigma = 3$ and noise with standard deviation of 0.001.

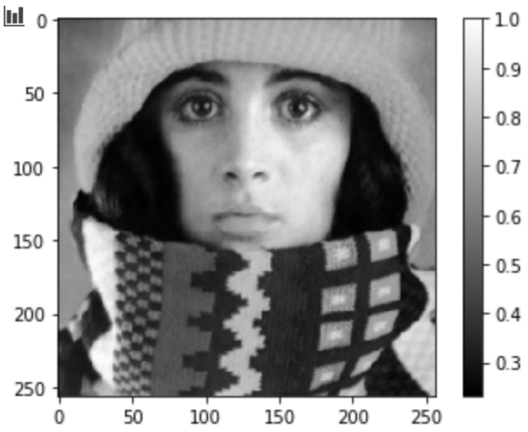
Figure 11: Results of direct inverse filtering with different strengths of noise.

In Figure 11 we see the results of applying direct inverse filtering on the LSI degraded image where we have blurred the image with a Gaussian kernel with $\sigma = 3$. We simply take the Fourier transform of the LSI degraded image and the psf and then divide the image with the psf. As we see, when there are no noise present the restoration results are very good, and all the blurring disappears. However, when we add just a tiny bit of noise it can be seen in the result. Already when we add noise with a standard deviation of 0.00000001 the results becomes almost useless, and when we further increase the noise we simply get garbage. Thus we conclude that the direct inverse filtering approach works very well when there are no noise, but really quickly we get garbage if noise is present. In Figure 12 we see the code used for this exercise.

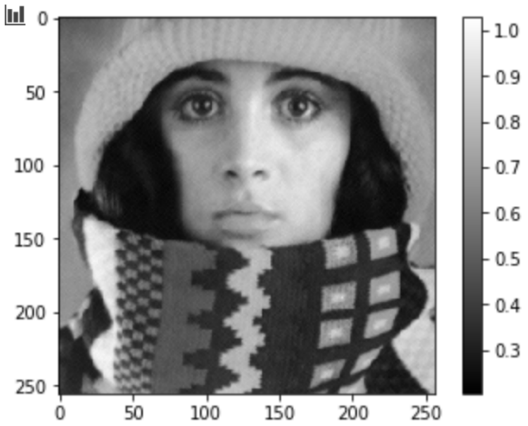
```
def inverse_filtering(im,psf):
    sz = (im.shape[0] - psf.shape[0], im.shape[1] - psf.shape[1])
    psf = np.pad(psf, (((sz[0]+1)//2, sz[0]//2), ((sz[1]+1)//2,sz[1]//2)), mode='constant')
    psf = ifftshift(psf)
    psf = fft2(psf)
    fft = fft2(im)
    fft_filtered = fft / psf
    f = ifft2(fft_filtered)
    return np.real(f)
```

Figure 12: Code used for this exercise.

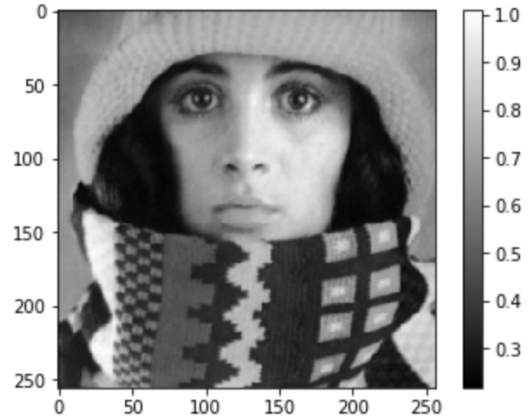
5.3



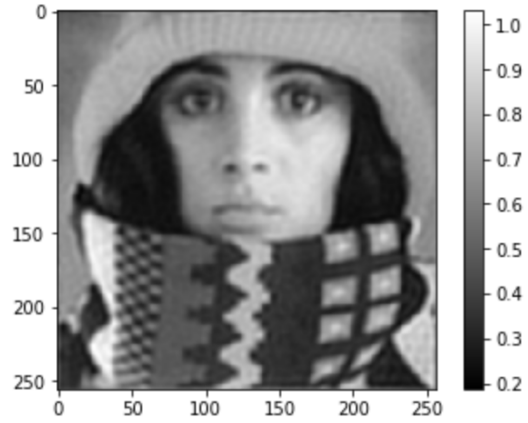
(a) Gaussian kernel with $\sigma = 3$, no noise and $k = 0$.



(b) Gaussian kernel with $\sigma = 3$, noise with standard deviation of 0.000000002 and $k = 0.000000000000001$.



(c) Gaussian kernel with $\sigma = 3$, noise with standard deviation of 0.00000001 and $k = 0.0000000000005$.



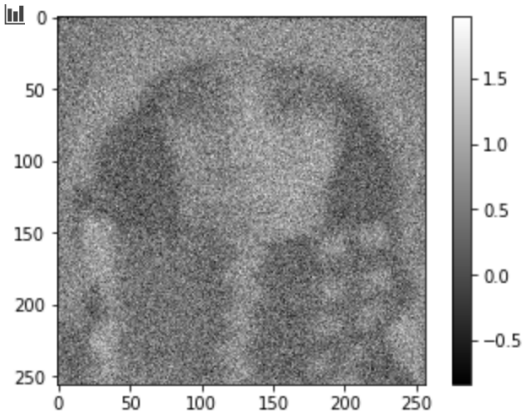
(d) Gaussian kernel with $\sigma = 3$, noise with standard deviation of 0.001 and $k = 0.0005$.

Figure 13: Results of Wiener filtering with different strengths of noise.

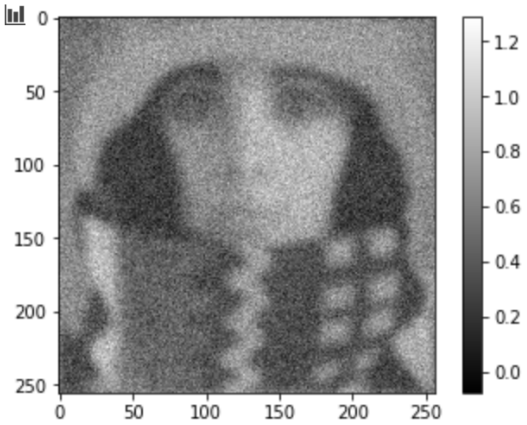
In Figure 13 we see the results of the Wiener filter on the same images as in the previous exercise. We here see a clear performance increase as the first 3 images are restored to perfection and the last is only slightly blurred still. In Figure 14 we see some examples of Wiener filtering on images with a little more extreme noise and blurring. In the first image we see an image almost completely gone in noise, and although the Weiener filter can not remove neither the noise or blurring completely, we still see a silhouette of the trui image with a clear scarf pattern, an indication of mouth and nose and blurred eye sockets. The second image is not completely gone in noise, but still heavily covered, and the result is a more clear image with both blurring and noise, but not disturbingly much noise. In additions to the features we could see in the previous image we can now clearly see the eyes, nostrils and eyebrows. In the last image the noise has been tuned down, but the blurring has been increased. The result is the most clear of the three, but still covered in both noise and blurring, but the restored image is clear.

During the experimentation with k values, noise and blurring I note that to decrease the blurring we want a small k value, whereas to remove noise, we want a high k value. Because of this, a lot of the restored image has been a compromise between removing enough noise to see the underlying image, but also removing the blurring effect.

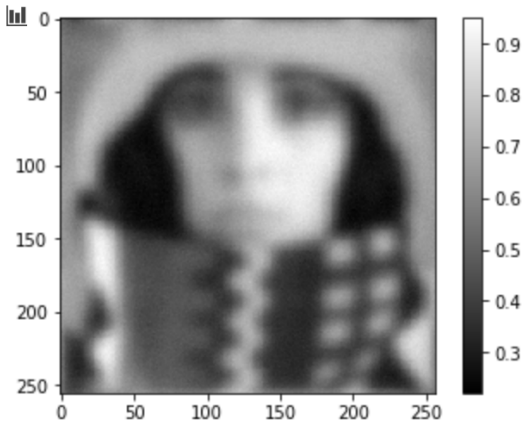
We see that the Wiener filter can to a high degree restore the image to a clear state where most features can be seen clearly if we apply low to medium noise, and can it can restore a fairly good silhouette of the image when the noise becomes high. Thus its abilities to restore images are good.
The code for this exercise can be seen in Figure 15. We here take the Fourier transform of the image and the psf and implement eq. 5.8-6 from Gonzales and Woods.



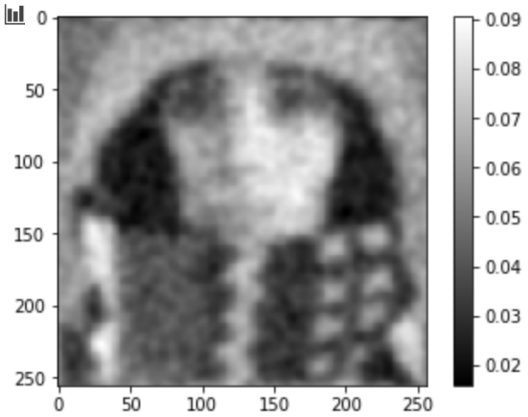
(a) LSI degraded image with Gaussian kernel with $\sigma = 3$ and noise with standard deviation of 0.3 .



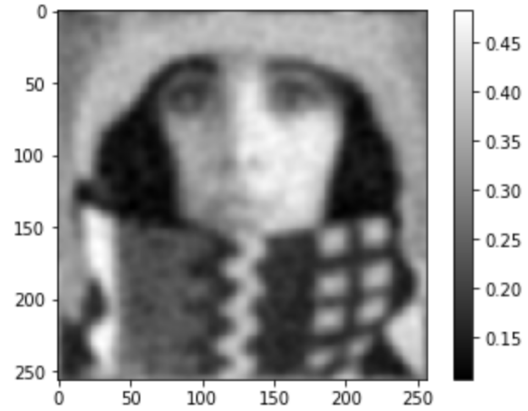
(c) LSI degraded image with Gaussian kernel with $\sigma = 3$ and noise with standard deviation of 0.1.



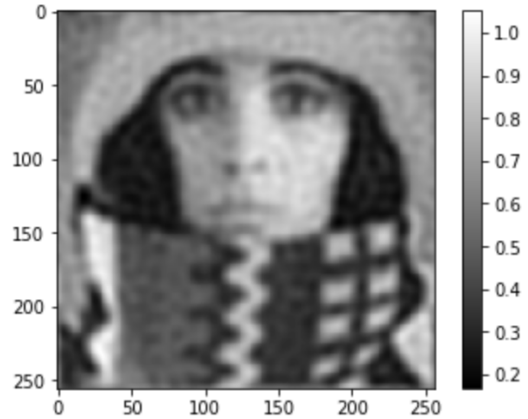
(e) LSI degraded image with Gaussian kernel with $\sigma = 5$ and noise with standard deviation of 0.01.



(b) Wiener result on LSI degraded image with Gaussian kernel with $\sigma = 3$, noise with standard deviation of 0.3 and $k = 10$.



(d) Wiener result on LSI degraded image with Gaussian kernel with $\sigma = 3$, noise with standard deviation of 0.1 and $k = 1$.



(f) Wiener result on LSI degraded image with Gaussian kernel with $\sigma = 5$, noise with standard deviation of 0.01 and $k = 0.002$.

Figure 14: Results of Wiener filtering with different strengths of noise.

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

Figure 15: Code used for this exercise.