Signal and image processing
Assignment 5

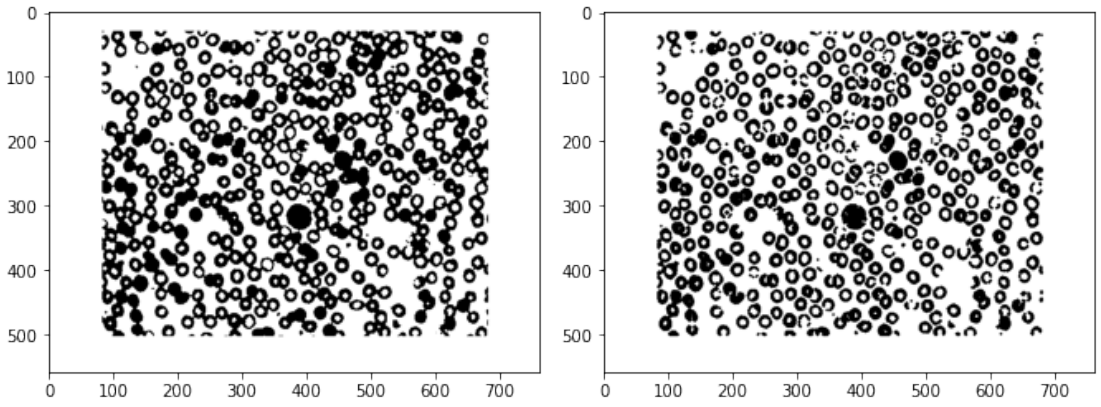
Marcus Hansen - tkz347
Casper Bresdahl - Whs715

March 14, 2021

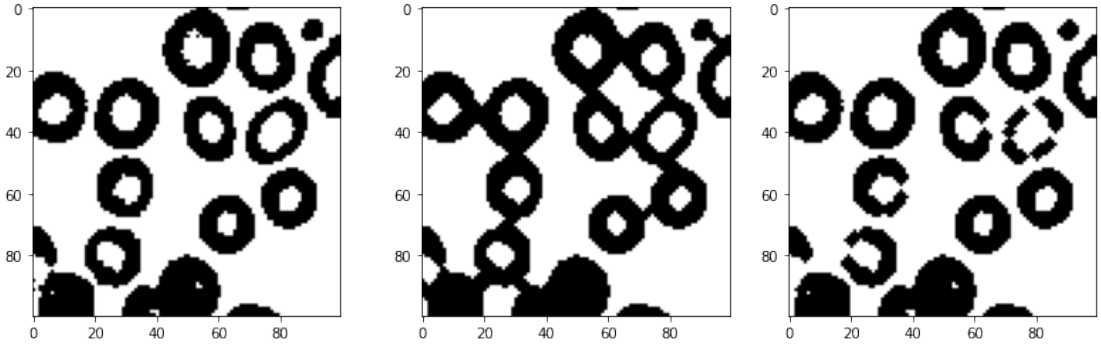
Exercise 1

1.1

1.1.1



(a) Opening (b) Closing



(c) Unchanged (d) Opening (e) Closing

Figure 1: Examples of performing morphological opening and closing we a ball SE with a radius of 2

We see that when performing opening that small white regions get filled out. This results in blobs close together becoming connected to each other, while the circular regions inside the blobs become more smoothed, or filled out for smaller blobs. We see the opposite for the closing operation. Here we see thin borders in the blobs get broken up while the outer shell of the blobs become more smoothed out.

1.1.2

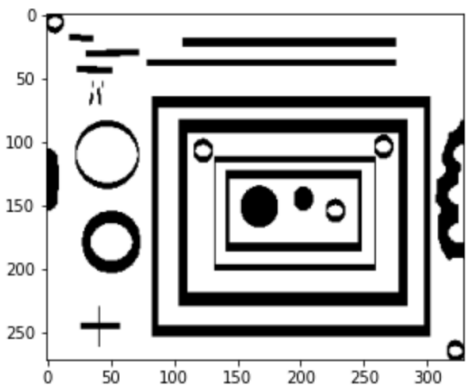
The two operations produce two different results. Since the opening operation is an erosion and then dilation $((A \ominus B) \oplus B)$ and closing is a dialation and then erosion $((A \oplus B) \ominus B)$ that means that for opening we might remove small areas that will not be recovered, and for closing it means that we might deform areas causing the breaks we see above.

When trying to segment with only the opening method, we will start to run into issue since cells will start to merge, causing it to become hard for us to differentiate between cells, with other morphological operations. When only using the closing method, we risk breaking up cells into many fragments that later might get identified as a cell which further breaks our segmentation.

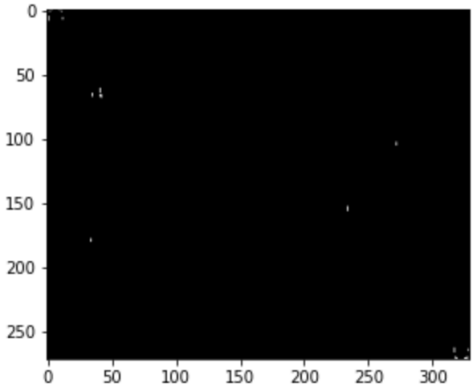
Other morphological operations that are useful would be, skeletonization and thinning, which help reduce objects to their root forms. Hit-or-miss which helps with pattern mathing and therein segmenting out certain patterns.

1.2

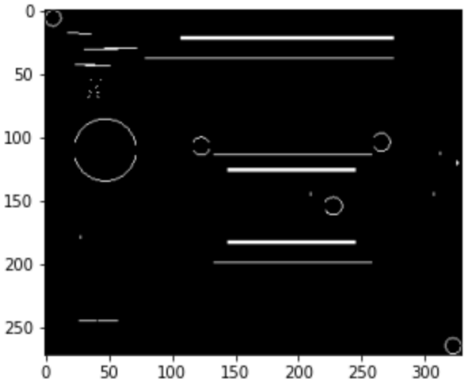
In Figure 2, Figure 3 and Figure 4 we see the results of applying the operations on respectively SE 1, SE 2 and SE 3.



(a) Hit-or-Miss with SE 1.



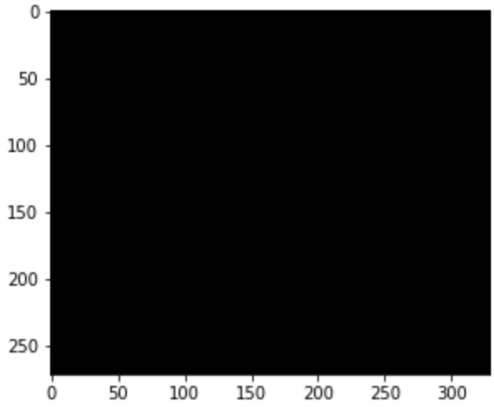
(b) TopHat with SE 1.



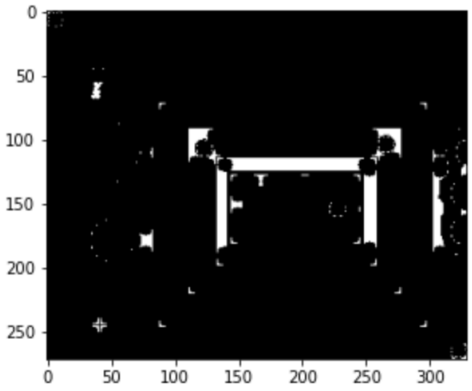
(c) BottomHat with SE 1.

Figure 2: Applying the operations with SE 1.

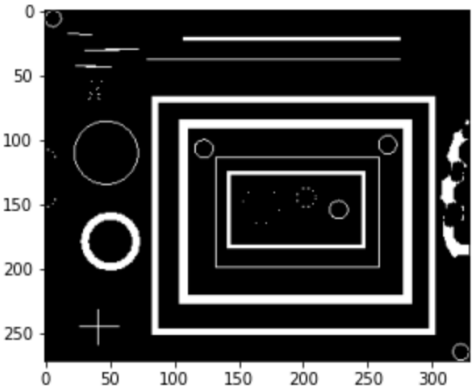
When looking at SE 1, we see the hit-or-miss operator pattern matches the vertical line in the image, and we get almost the same image back, except the dark horizontal lines has grown a little. In the top hat we subtract the opened image from the original image but we do not seem to pick up any specific features. In the bottom hat we subtract the original image from the closed one, and here we see a response on horizontal lines.



(a) Hit-or-Miss with SE 2.



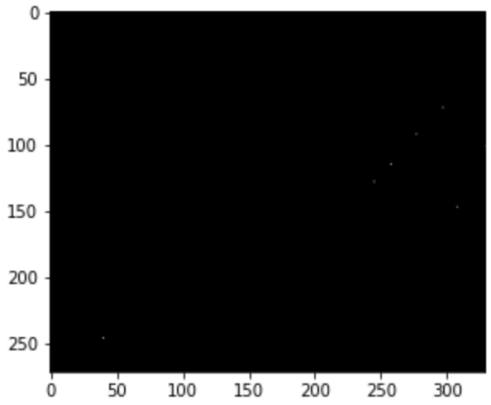
(b) TopHat with SE 2.



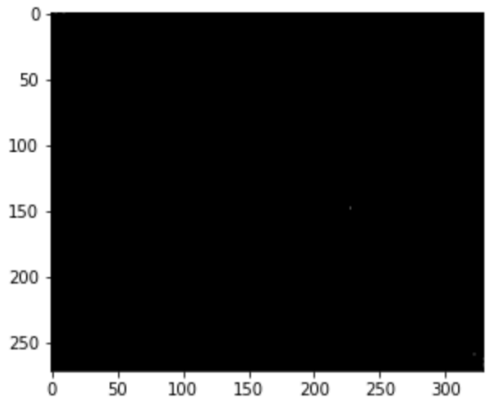
(c) BottomHat with SE 2.

Figure 3: Applying the operations with SE 2.

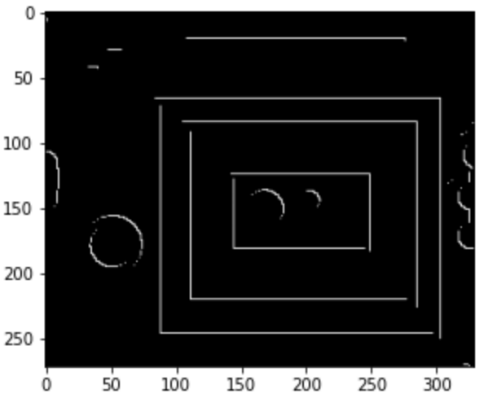
Considering SE 2, we do not seem to be able to pattern match the disk with the hit-or-miss operation, and thus a black image. With the top hat we seem to pick up some corners and thick straight lines whereas the thinner lines does not yield a response. With the bottom hat we seem to get the image back more or less untouched. Some of the circles and the structures in the left and right of the images seem to have been thinned a little. We thus seem to pick up on features consisting of line segments surrounded by background with the bottom hat.



(a) Hit-or-Miss with SE 3.



(b) TopHat with SE 3.



(c) BottomHat with SE 3.

Figure 4: Applying the operations with SE 3.

Considering SE 3, we pattern match on top right corners with the hit-or-miss operator, and we see a few matches scattered around the image. With the top hat we seem to have a single match, but not really a specific feature. With the bottom hat we seem to pick up on the top right corner feature again, although we get a lot more structure together with it. We can for instance see the whole square, and the outline of the circles.

1.3

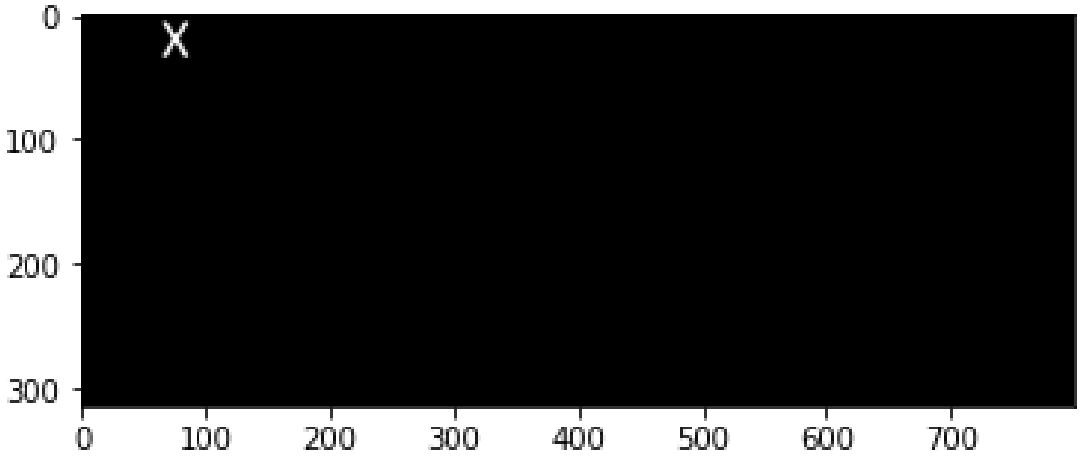
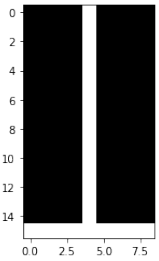
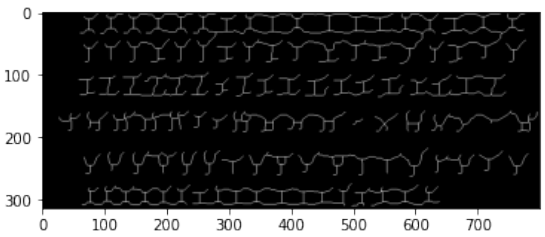


Figure 5: First method of pattern matching X's

We only get one hit when applying hit-or-miss with the x. This is due to there only existing one single x of that shape and size. Therefore to improve our matching we need to have a mask that is represented by more than one x. But also is only represented there and not other places. Therefore a mask that is a skeletonize version of an x should perform better. But this also requires that the image we are matching on is also skeletonize, as otherwise we would get 0 matches most likely. This is further improved by iteratively dilating and skeletonization of the image, as this slowly forces the different digits to conform to a similar skeleton structure. Finally hit-or-miss is used to find the matching digits.



(a) Modified mask



(b) m̈oney$_b$*in.jpgäftermodification*

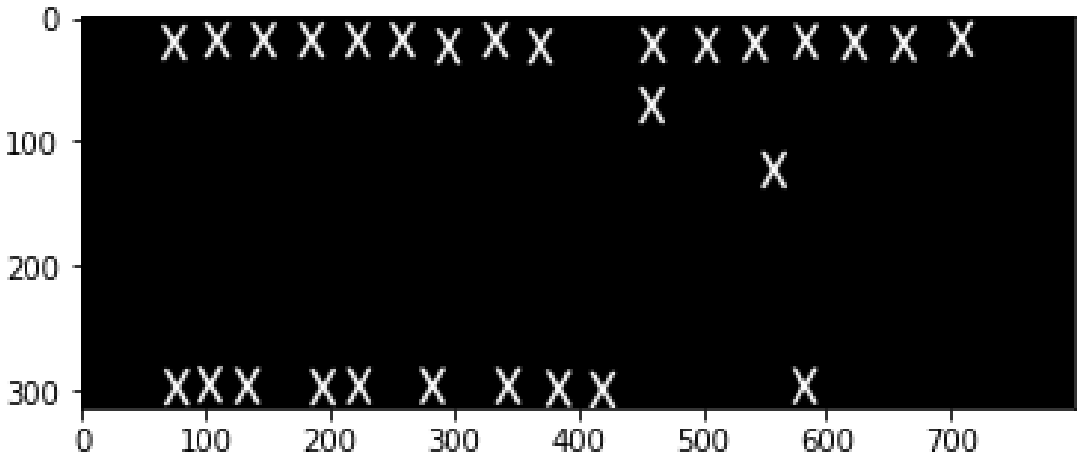
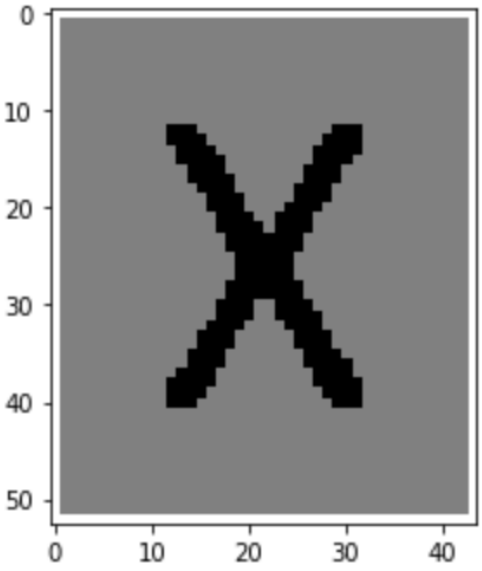


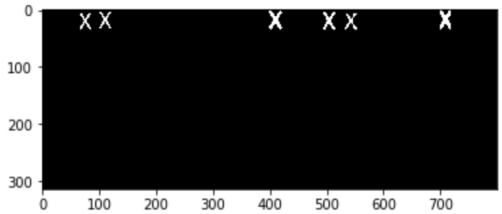
Figure 7: Modified method of finding matches

0.0.1 Extra method

As we are unsure whether we are allowed to change the letter image, we also came up with another strategy which proved less efficient however. We constructed a new x to match on which can be seen in Figure 8a which uses -1 for part of the x, 1 for boundary and 0 everywhere else. 0 in this context signals we do not care about the pixel value, this way, we only care about the x being on some white background and the middle of the x having an x shape. The results can be seen in Figure 8b.



(a) Code used to count clusters.

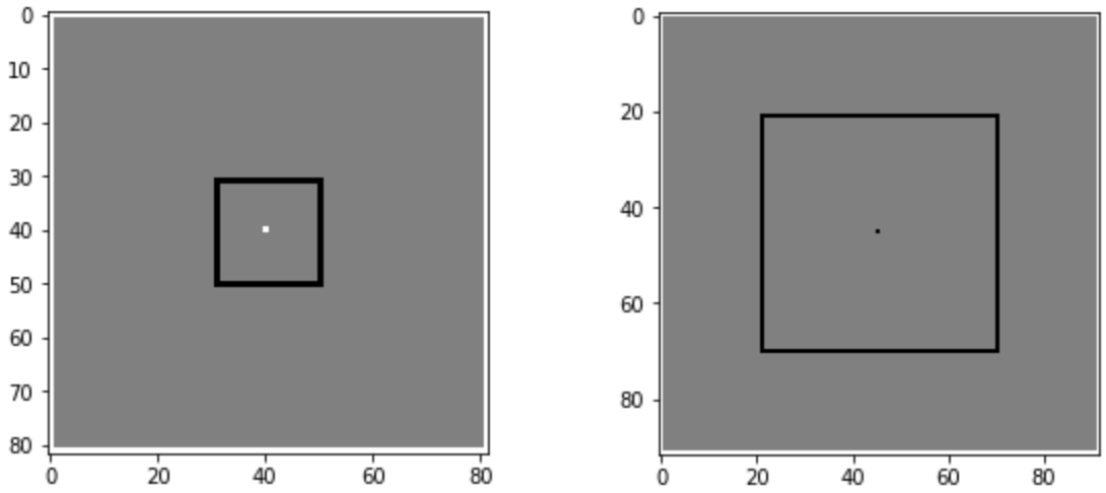


(b) Result of cv2.morphologyEx with cv.MORPH_HITMISS. The hits are dilated with the original x cut-out.

Figure 8: Mask and result of second approach.

1.4

To solve this exercise we first created several masks like the ones seen in Figure 9. These vary in overall size, and in the size of the black square, and whether it has a white or black dot in the middle. We also note the white 1 pixel border around the image. Using the hit-or-miss operator we pattern match each of these masks in the original image. Each mask resembles the approximate size and characteristics of a coin, and after applying the hit-or-miss operation, we thus get a cluster of white pixels at the location of the coin resembling the mask. We get a cluster because the mask is not perfect, and can be matched several places inside the coins. In turn we apply the hit-or-miss operation with the masks, and we count the clusters in the resulting image, multiplying by the value of the mask / coin we thus find the sum of money to be 115.5 kr. In Figure 10 we see the main code used for this exercise, and in Figure 11 we see the code used to count the clusters.



(a) Mask used for 2kr coin.

(b) Mask used for 20kr coin.

Figure 9: Two masks used for pattern matching.

```
def count_coins(im):
    s = 0
    masks = [m05,m1,m2,m5,m20]
    values = [0.5,1,2,5,20]
    mask_values = zip(masks,values)

    for m,v in mask_values:
        out = cv2.morphologyEx(A4, cv2.MORPH_HITMISS, m)
        n = count_regions(out, 20)
        s += n*v
    return s
```

Figure 10: Main code used for exercise 1.4.

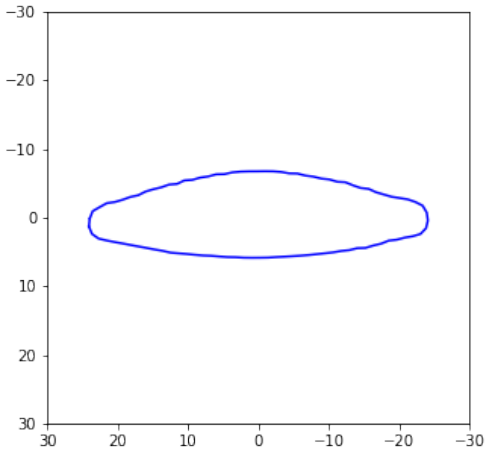
```python
def count_regions(mod, radius, foreground = 1):
    mod = np.array(mod)
    x_max = mod.shape[0]
    y_max = mod.shape[1]
    s = 0
    blobs = np.argwhere(mod == foreground)
    while len(blobs) > 0:
        x,y = blobs[0]
        mod[max(0,x-radius):min(x+radius, x_max),max(0,y-radius):min(y+radius, y_max)] = 0
        s += 1
        blobs = np.argwhere(mod == foreground)
    return s
```

Figure 11: Code used to count clusters.

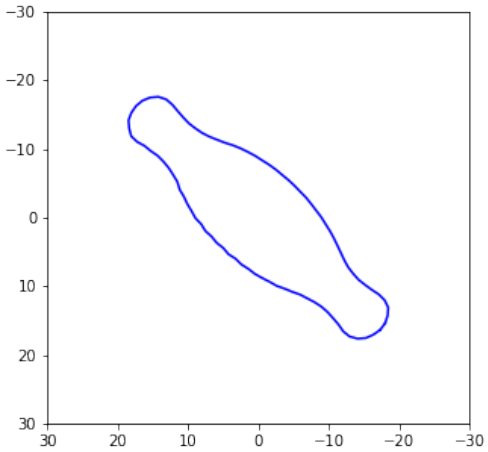
Exercise 2

2.1

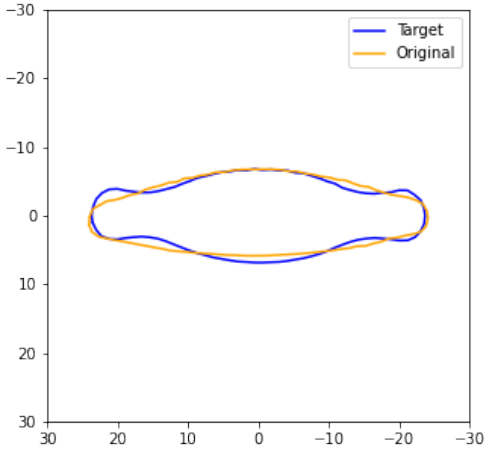
In Figure 12 we see the results of our Procrustes algorithm, and in Figure 13 we see our code.



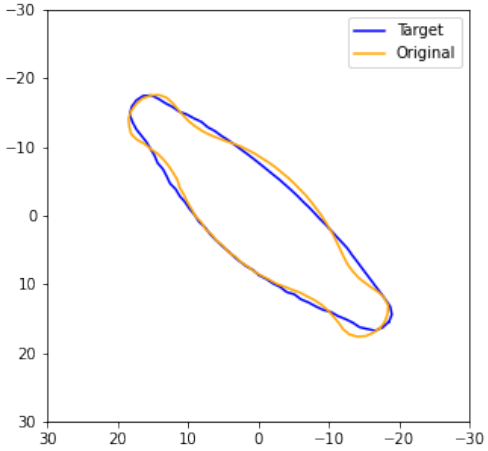
(a) The first diatom in the training set



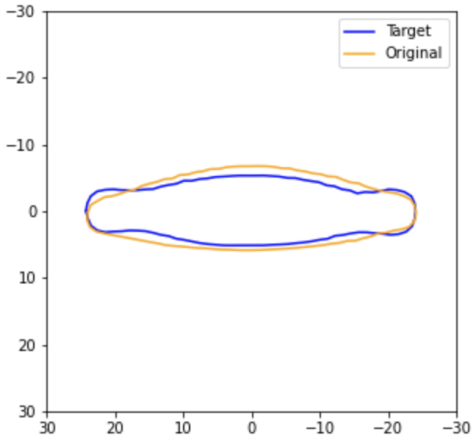
(b) Diatom 150 in the training set



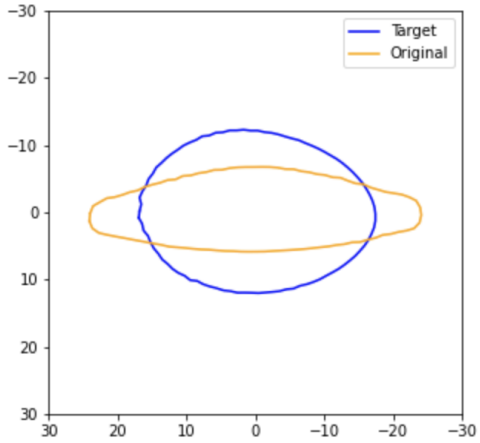
(c) The other diatom transformed onto the first diatom



(d) The first diatom transformed onto the other diatom



(e) Diatom 75 aligned.



(f) Diatom 36 aligned.

Figure 12: Before and after images of applying Procrustes

```python
def procrustes(diatom_a, diatom_b):
    def sum_vecs(vecs_a, vecs_b):
        s = 0
        for y, x in zip(vecs_b, vecs_a):
            s += y.T@x
        return s

    yx = diatom_a.T @ diatom_b
    u, s, v = np.linalg.svd(yx)
    R = v @ u

    b = diatom_b@R

    s = sum_vecs(diatom_a, b)/sum_vecs(b, b)
    scale_matrix = np.identity(2)*s

    return b@scale_matrix
```

Figure 13: The code implementing Procrustes.

2.2

Simply using the code from the appendix we train our classifier on the training data before aligning the diatoms, and then use the trained classifier to predict the test data gives us an accuracy of 3.2%. Training a new model using the aligned training data and then using this trained model to predict the test data we achieve 17.9% accuracy. We thus see a huge increase in accuracy simply by aligning our data.

2.3

When we align the diatoms, we change the scale and rotation of the diatoms which makes all the images look much more alike. The reason they look a lot more alike is because we remove the variance in the images. This variance also makes it harder for a classifier to learn which features to focus on. This is why when we remove the variance in the images, the classifier has a much easier time focusing on the shape rather than size or rotation, and thus we get much better prediction results. The information we keep after the alignment is the shape of the diatoms.

Exercise 3

3.1

A translation to the right is results in all pixels moving one pixel to the right which we can express as:

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

3.2

This translation can be expressed as:

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

In homogenous coordinates.
We can use this matrix to derive:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where x and y are coordinates in I and x' and y' becomes the coordinates in $\tilde{I}$.

3.3

A linear filter which shifts all pixels in an image by one pixel to the right after convolving with it is:

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

3.4

We can define a numpy array with a white center dot which looks like:

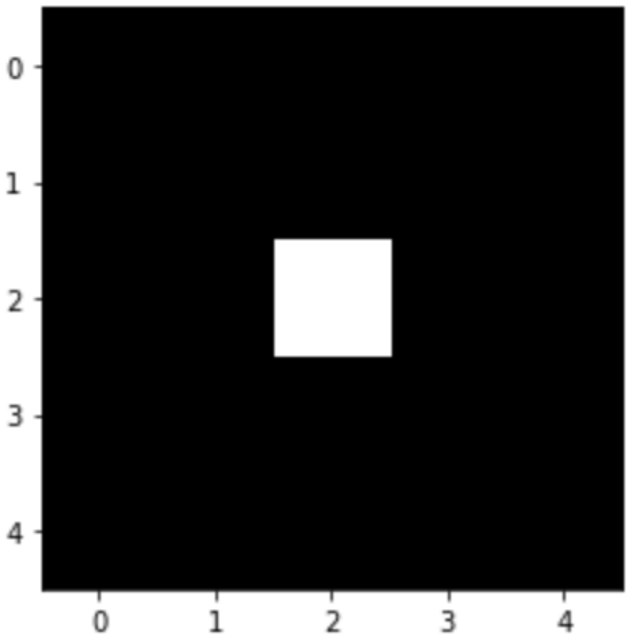


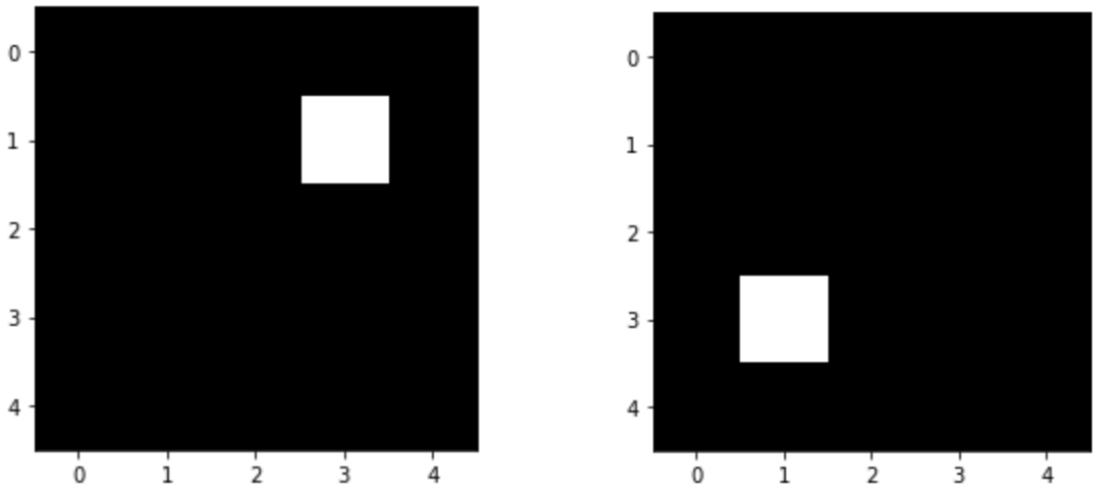
Figure 14: Odd sized image with white dot in center.

3.5

```
def shiftKernel(im, a, b):
    n = 2*max(abs(a),abs(b))+1
    k = np.full((n,n), 0)
    k[-b+n//2,a+n//2] = 1
    return scipy.signal.convolve2d(im, k, mode='same')
```

Figure 15: Code for exercise 3.5

In Figure 15 we see the code for this exercise. The function takes an image, a shift in the x direction and a shift in the y direction. The function has no boundary conditions, and giving a shift coordinates which moves the motive out of bounds simply gives back a black image. One could implement wrapping such that when the motive moves out of bounds, it wraps to the other side. In Figure 16 we see the results of shifting the white dot image.



(a) Shift of 1 to the right and 1 up.

(b) Shift of 1 to the left and 1 down.

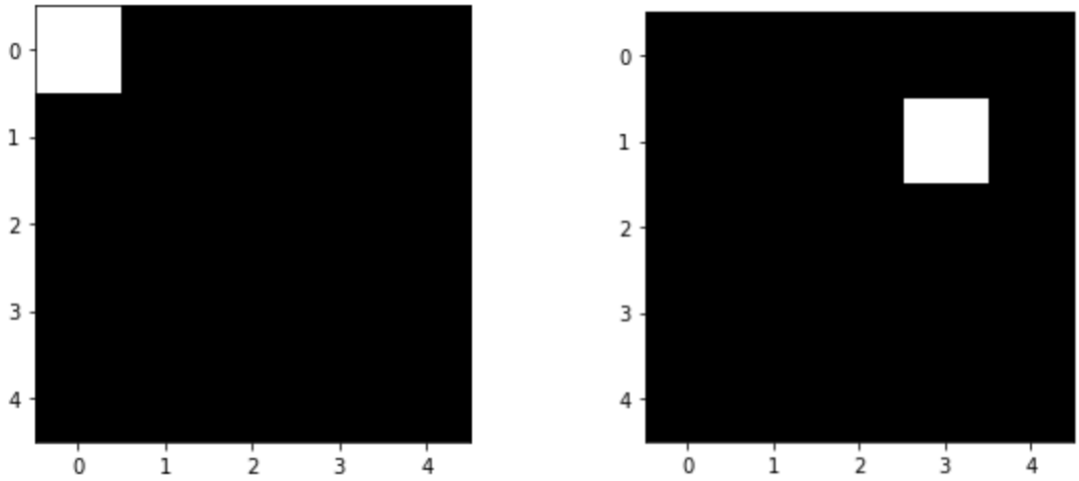
Figure 16: Results of shifting the white dot image.

3.6

```python
def homoTransform(im, tx, ty):
    c = np.full_like(im, 0)
    H = np.array([[1,0,tx],
                  [0,1,-ty],
                  [0,0,1 ]])
    for y in range(im.shape[0]-1):
        for x in range(im.shape[1]-1):
            v = np.array([[x],
                          [y],
                          [1]])
            nv = np.dot(H,v)
            nx = int(np.round(nv[0]))
            ny = int(np.round(nv[1]))
            if(nx < 0 or nx > (im.shape[1]-1) or ny < 0 or ny > (im.shape[0]-1)):
                continue
            c[ny,nx] = im[y,x]
    return c
```

Figure 17: Code for exercise 3.6

In Figure 17 we see the code for this exercise. The function takes an image, a shift in the x direction and a shift in the y direction. The function has no boundary conditions, and giving a shift coordinates which moves the motive out of bounds simply gives back a black image. One could implement wrapping such that when the motive moves out of bounds, it wraps to the other side. In Figure 18 we see the results of shifting the white dot image.



(a) Shift of 2 to the left and 2 up.

(b) Shift of 0.6 to the right and 1.2 up.

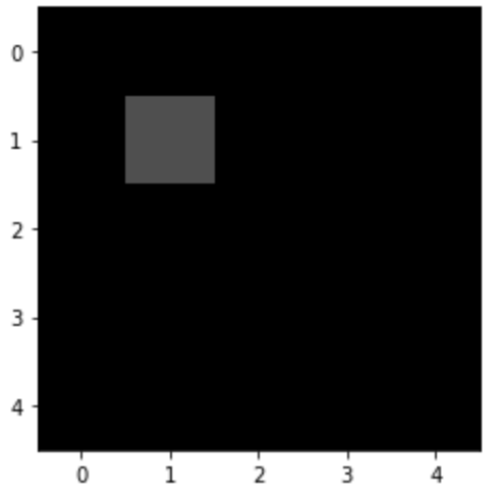
Figure 18: Results of shifting the white dot image.

3.7

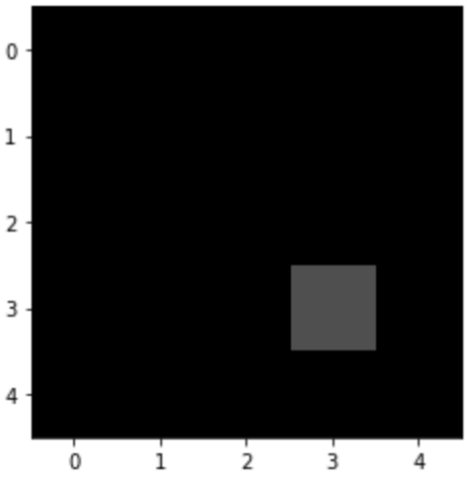
```
def fourier_shift(im, a, b):
    n1 = im.shape[0]
    n2 = im.shape[1]
    u,v = np.mgrid[0:n1,0:n2]
    k = np.exp(-2j*np.pi*(u*a/n1 + v*b/n2))

    k = fftshift(k)

    fft = fft2(im)
    filtered = fft * k
    return np.absolute(np.real(ifft2(filtered)))
```

Figure 19: Code for exercise 3.7

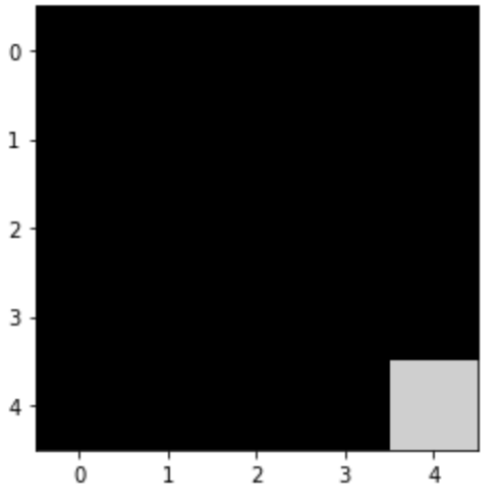
In Figure 19 we see the code for this exercise. The function takes an image, a shift in the x direction and a shift in the y direction. When comparing this approach to the others, we see that the intensity of the white dot changes as it shift away from the center, and especially when it not shifted to the corners. The results can be seen in Figure 20



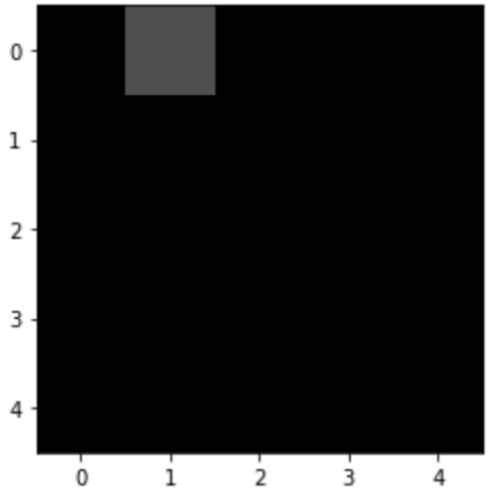
(a) Shift of 1 to the left and 1 up.



(b) Shift of 1 to the right and 1 down.



(c) Shift of 2 to the right and 2 down.

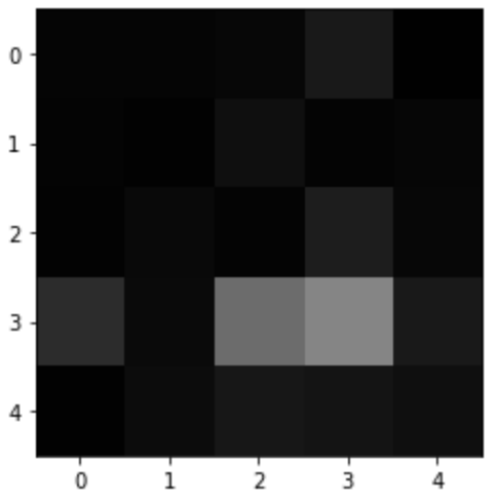


(d) Shift of 1 to the left and 2 up.

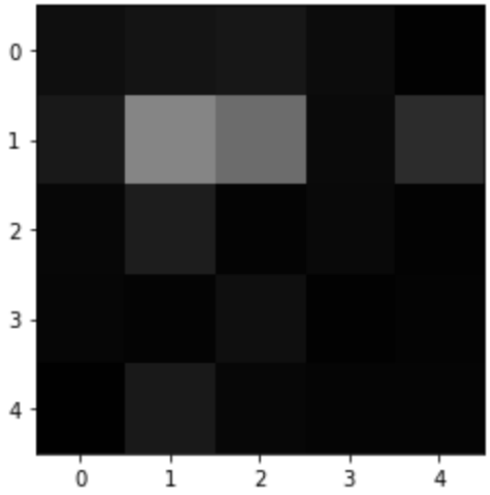
Figure 20: Results of shifting the white dot image.

3.8

You can do non-integer translation with the Fourier method. When shifting with floats in the Fourier transform method, the white dot gets split into several pixels. The brightest part seems to be the position we actually shift to. In Figure 21 we see the results for the white dot image.



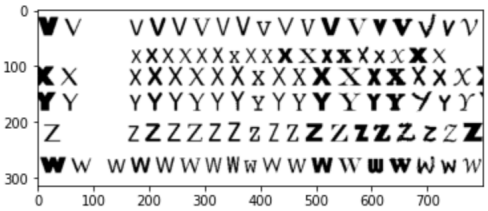
(a) Shift of 0.6 to the right and 1.2 down.



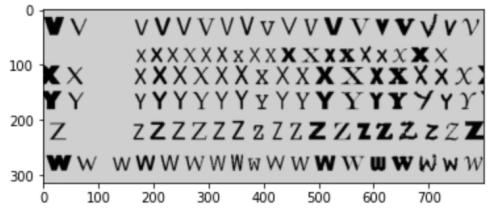
(b) Shift of 0.6 to the left and 1.2 up.

Figure 21: Results of shifting the white dot image.

If we shift the digit image using the Fourier method we get Figure 22a and if we shift by a float we get Figure 22b. Where we see the float shift results in a 'light dim' effect in the image.



(a) Shift of 100 to the right and 100 down.



(b) Shift of 100.9 to the right and 100.9 down.

Figure 22: Results of shifting the digit image.