University of Copenhagen

Department of Computer Science

Signal Image Processing (SIP)

Assignment 5

Bjarke Heldbo, Toke Reines & Frederik Christiansen
ku-id:
rvs299 , bkx591 & nzt365

Contents

14. marts 2022

Version: 22:07:55 14 pages

1 Morphology

1.1 Opening and Closing

1.1.1 Highlighted key locations

First off, observe, that in contrary to the lecture video, `morphology.mp4`, the shapes in this image are black on white background, whereas the shapes in the lecture video are white with black background. Because of this, the behavior of `Opening` and `Closing` will be swapped.
From the image below, we observe 3 areas of significant changes:

The red area shows loss of data when opening (closing in this case) - notice the dot in the original image persist in the Opened image, but disappear in the Closed image. The Closed image is actually opened, which is the act of erosion followed by dilation, where the dot is removed when eroding.

The blue area shows what happens to objects with "holes" in them. Notice the cell in the bottom right of the blue area being opened (closed in this case) and therefore changing shape from a cell to a grainy horse-shoe.

The green area shows the largest cell in the image being filled with color (black) when closing (opened in this case), which is the same behavior we observe in the lecture video on absalon on the bottle-opener shape at 52:00 where the hole gets filled to match the rectangular shape.

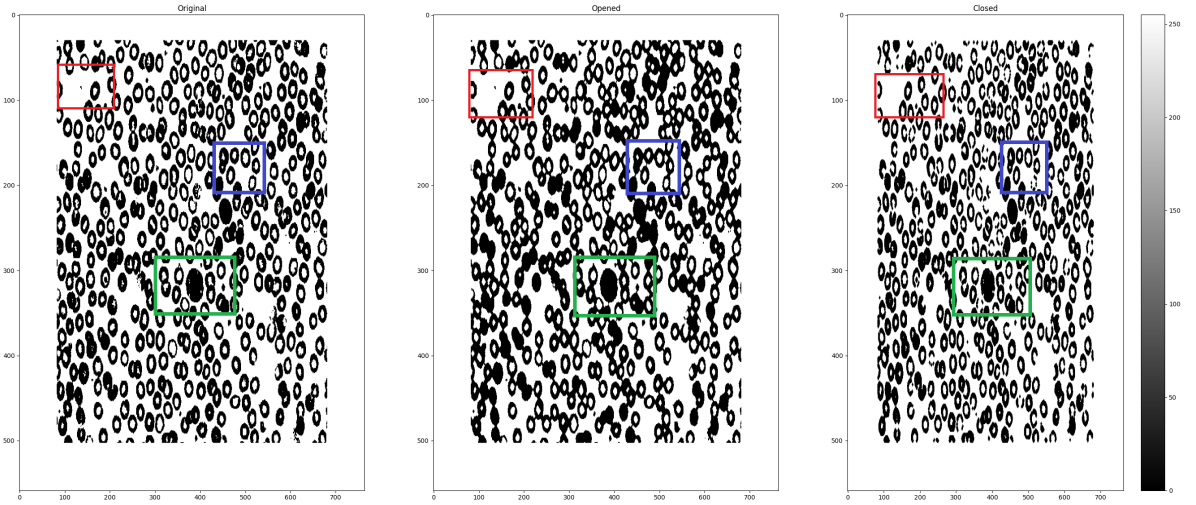

Figure 1: Performing Opening and Closing using skimage

1.1.2 Questions

i The two operations are not producing the same result due to the order of sub-operations (dilation/erosion). Opening does erosion followed by dilation and Closing does dilation followed by erosion - you can remember the order of operations by noticing that Opening starts with a vowel, as does erosion, and Closing starts with a consonant, as does dilation.

ii One of the challenges of using just these two operations for segmenting this particular image into individual cells is, that the cells are overlapping each other a lot. Each cell is a circle with a hollow core, which could be segmented by Opening and Closing if they did not overlap but only touched each others borders. In short, we will never be able to isolate overlapping cells from each other, but merely erode or dilate overlapping cells until they have "merged" into one cell.

iii Other morphological operations that could be useful for segmenting these cells are; skeletons - in hopes of finding a unique skeleton per cell, and also ultimate erosion for cell-center finding (maybe we could performing closings beforehand to fill up cells). Watershedding might also be an option,

where every basin would represent a cell.

1.2 Top/Bottom hat and Hit and Miss

The tophat (white) is the difference between the original image and the resulting image from performing opening with a given SE. The bottomhat (black) is the difference between the resulting image from performing closing with a given SE and the original image. We know, that closing and opening can be interchanged depending on the foreground/background color, which is why we now refer to them as white/black tophats. Top/Bottom hat operations are good tools for extracting objects on backgrounds of a contrasting color.

1.2.1 White tophat

In the White tophat image, we should see bright (white) objects that are smaller than the SE - which seems to be only noise in this case. This also seems to be the case for all SEs and White tophat images, except the 5x5 disk where it seems, that all inner corners are visible (notice the 4 inner corners in the cross in the bottom left.

1.2.2 Black tophat

More information is extracted in the black tophat, where the Corner SE (right side) extracts all vertical, horizontal, and upper right-most edges. The vertical line SE (left side) extracts all horizontal lines whose width is less than 5 pixels (height of SE). The disk (middle) works in a similar way to the vertical line, except that it also extracts vertical lines with a width less than 5 pixels

1.2.3 Hit or miss

The Hit or miss results shows, that the vertical bar highlights everything as in the original image. This is due to the implementation and situation with a SE with only 1's. The 5x5 disk looks for an exact disk in the image, which is not present - therefore a black image. The corner SE is the most interesting one. It finds all the upper-right corners, which gives us a dot for all the 4 rectangles, the bottom-right cross and then a dot in the middle-right due to the tacky shape in the outmost right side of the image.

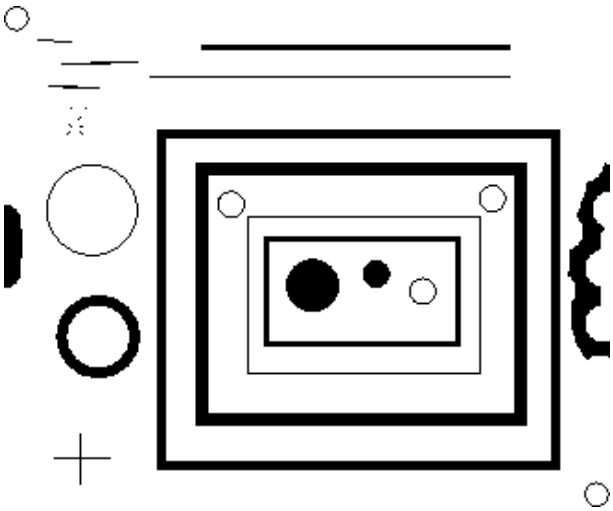

Figure 2: Original image, blobs_inv.png

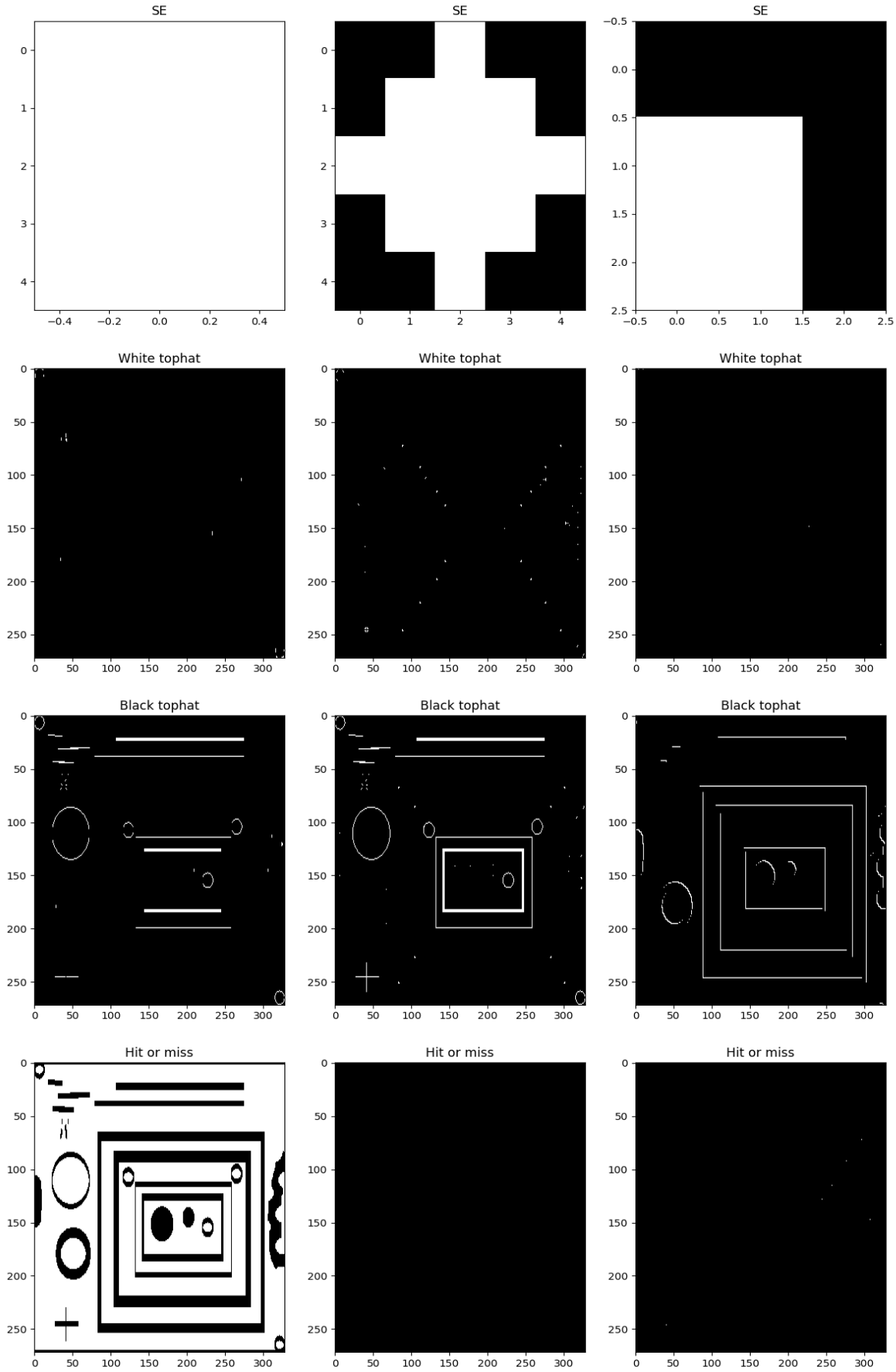

Figure 3: Results from performing top/bottom hat and Hit-or-Miss with different SEs

```
image = np.array(imread("cells_binary.png", as_gray=True)).astype('uint8')
image[image > 0] = 1
vertical_line = np.array([[1],
                          [1],
                          [1],
                          [1],
                          [1]])
disk = morphology.disk(2)  # Gives a 5x5 SE
corner = np.array([[0, 0, 0],
                   [1, 1, 0],
                   [1, 1, 0]])

for i, SE in enumerate([vertical_line, disk, corner]):
    white_tophat = morphology.white_tophat(image, SE)
    black_tophat = morphology.black_tophat(image, SE)
    hit_or_miss = binary_hit_or_miss(image, structure1=SE).astype('uint8')
```

1.3 Hit or Miss Coins

Using the first X from the first line in the image `digits_binary_inv.png` to find other X's of similar shape, we had to do the following: First we binarized the image with a threshold of 50, then we cutout the X as a template/kernel and then we applied `Hit or Miss` on the binarized image itself. This gives us exactly 1 hit as shown in figure 4. Python code for the `Hit or Miss` functions are also shown. We get this result, as the `Regular Hit or Miss` involves two erosion steps. One on the image with the mask and one with the complementary image and the complementary mask. The first step, erosion with a black x on an image with black letters, gives an almost pure black image (background in erosion), as the X stretches many pixels (20+ pixels) for a mask, so the minimum value for "on"-pixels for every erosion step will most likely be 0. The inverse of this, step 2, means eroding a black image with white letters, with a black mask with a white X. The minimum value will only be 0 when the white ("on"-pixels) X is completely engulfed in black pixels - which happens to be on larger X's that can contain our mask X. Applying the logical and operator to our two resulting images will give us only the exact match, but using only step 2, the complementary erosion, gives us a relaxed `Hit or Miss` functionality as shown in figure 5. Our comments on the limitations on this approach is, that we are totally dependent on correct rotation of our mask and its potential matches in the image. We observe, that any X that can contain the mask X inside of it, will be a match, but smaller X's will not. So this is not a good approach for real images with depth - where objects seem smaller in the distance. Also the shape of X is not pure, as rotation, skew etc has not been corrected, as we have learned possible in the topic of transformations and shape. Finding the shape of our X, would likely distance it too much from other X's in the image, though, and then give us fewer matches.

```
def hit_or_miss(image, mask1):
    compl_mask = 1 - mask1
    compl_image = 1 - image

    step1 = binary_erosion(image, mask1)
    step2 = binary_erosion(compl_image, compl_mask)
    return np.logical_and(step1, step2)

def relaxed_hit_or_miss(image, mask1):
    compl_mask = 1 - mask1
    compl_image = 1 - image

    step2 = binary_erosion(compl_image, compl_mask)
    return step2
```

Listing 1: Regular and Relaxed Hit or Miss function

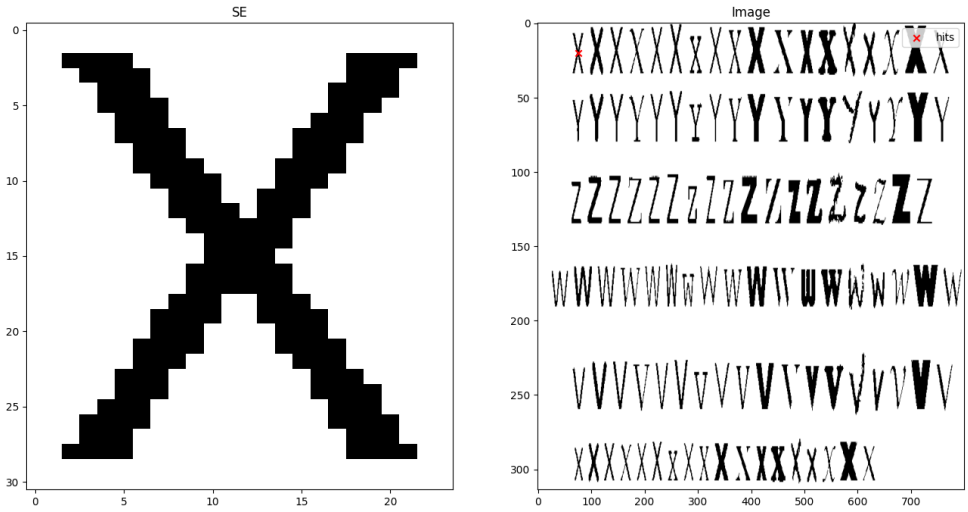

Figure 4: Results of Regular Hit or Miss with x as a kernel on the image `digits_binary_inv.png`

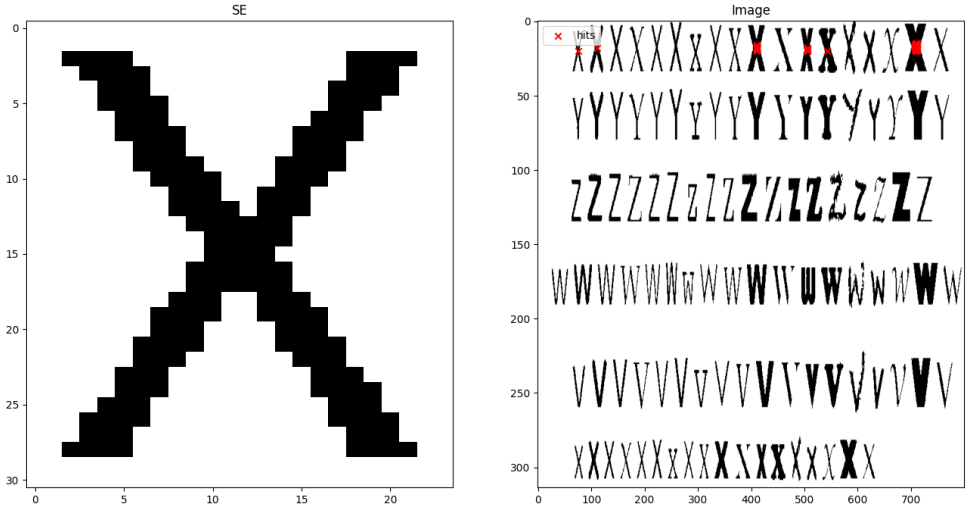

Figure 5: Results of Relaxed Hit or Miss with x as a kernel on the image `digits_binary_inv.png`

1.4 Money Counting

After spending much time attempting to count the money, we were unsuccessful. Please enlighten us as to how to do this when you are correcting this assignment. We attempted eroding the image with a small disk, to get all coins to increase a tiny bit in size. Then we tried using our relaxed Hit or Miss method to find Hits for every coin. This of course gave too many false positives, so we started searching in size-descending order, iteratively finding matches. For every hit, we tried to remove the coin from the image by using Connected Component labeling, to prevent false positives from smaller coins. We had troubles grouping Hits together, though, as 1 coin-mask could have 3 hits for the same coin. Using Connected Component labeling, we could group them, if the pixel-color-value they were positioned at, had the same color. Though we tried, our effort was not enough to count the coins. Our conclusion for now, is that there is at least 25 kr :).

2 Transformations on point clouds

2.1

Our implementation of Procrustes transformation is partly based on the slides `transformations2.pdf` p. 15 and inspiration from `scipy.spatial.procrustes`[1] to patchwork the following implementation:

```
def setup():
    X_train = np.loadtxt('diatom/SIPdiatomsTrain.txt', delimiter = ',')
    X_test = np.loadtxt('diatom/SIPdiatomsTest.txt', delimiter = ',')
    X_train_classes = np.loadtxt('diatom/SIPdiatomsTrain_classes.txt', delimiter = ',')
    X_test_classes = np.loadtxt('diatom/SIPdiatomsTest_classes.txt', delimiter = ',')
    return X_train, X_test, X_train_classes, X_test_classes

def procrustes(target, train):
    target_pro = np.copy(target)
    train_pro = np.copy(train)

    # Translational Alignment
    target_pro -= np.mean(target_pro, 0)
    train_pro -= np.mean(train_pro, 0)

    target_pro /= np.linalg.norm(target_pro)
    train_pro /= np.linalg.norm(train_pro)

    # Rotational Alignment
    U, s, V = np.linalg.svd(np.dot(train_pro.T, target_pro))
    R = np.dot(U, V.T)
    train_pro = np.dot(train_pro, R)

    # Scaling Alignment
    train_pro = (train_pro * s.sum())
    return target_pro, train_pro

def procrustes_tranform(dataset):
    X_train, _, _, _  = setup()

    target_points = np.stack((X_train[0, 0::2], X_train[0, 1::2]), axis=1)
    train_points = np.stack((dataset[0:, 0::2], dataset[0:, 1::2]), axis=2)

    # Procrustes takes a target, and an entry to be transformed
    pro = [procrustes(target_points, train_points[i]) for i in range(len(dataset))]

    pro = np.array(pro, dtype=object)

    # Unpack variables
    target_pro, train_pro = pro[:,0], pro[:,1]

    # We're interrested in train_pro for task 2.2, and return as same format and shape as dataset:
    return np.array([train_pro[i].flatten() for i in range(len(dataset))])
```

To see how we plot, see appendix (3).

[1] https://github.com/scipy/scipy/blob/v1.8.0/scipy/spatial/_procrustes.pyL15-L130

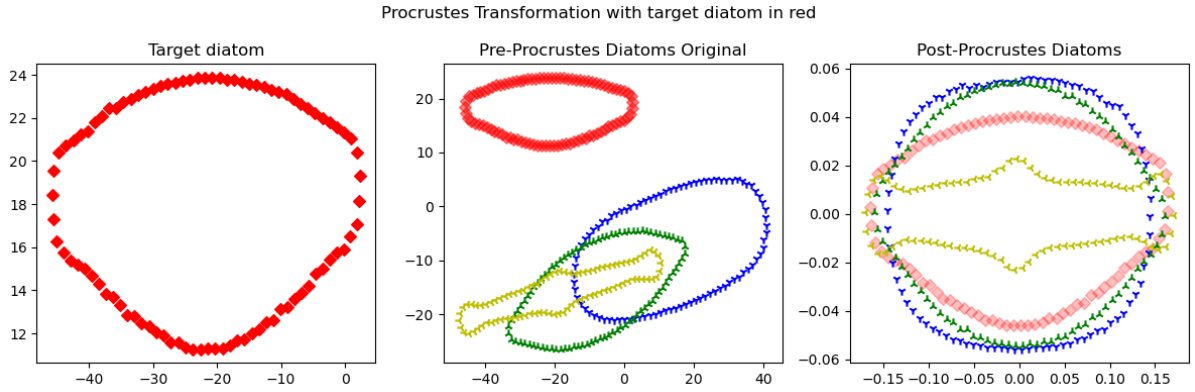

Figure 6: Procrustes transformation on diatom 2-4 in `SIPdiatomsTrain.txt`, w/ target diatom 1 in red

The leftmost plot shows the target diatom, drawn by its x and y coordinates given by the dataset `SIPdiatomsTrain.txt`, the target diatom is the first one in the training set (index: 0) and is drawn in red on all the plots. In the middle, other diatoms from the same dataset is shown in different colours; they are the diatoms (index: 1-3) following the target diatoms index in the dataset, respectively drawn in blue, green and yellow.
The target diatom is standardised to be plotted with the post-Procrustes diatoms in the rightmost plot.

2.2

Below is our implementation for task 2.2

```
def task2_2():
    # First fit an RF on train set with no Procrustes transformation
    clf = RandomForestClassifier(n_estimators=180)
    X_train, X_test, X_train_y, X_test_y = setup()
    clf.fit(X_train, X_train_y)
    # Predictions on the test set
    y_preds = clf.predict(X_test)
    acc_pre = clf.score(X_test, X_test_y)

    # Fit another RF on Procrustes transformed entries
    clf2 = RandomForestClassifier(n_estimators=180)
    X_train_procrustes = procrustes_tranform(X_train)
    X_test_procrusted = procrustes_tranform(X_test)
    clf2.fit(X_train_procrustes, X_train_y)
    # Predictions on the standardised test set
    y_preds = clf2.predict(X_test_procrusted)
    acc_post = clf2.score(X_test_procrusted, X_test_y)

    print("Accuracy for RFC on pre-Procrustes: %.4f" %acc_pre)
    print("Accuracy for RFC on post-Procrustes: %.4f" %acc_post)
```

As you can see, we went for random forest classifier. Various SKlearn library classifiers were tested where Random Forest Classifier(RF,RFC) were found to achieve the highest accuracy.
Using the function `procrustes_transform()` (shown in task 2.1), we fit the two classifiers, one with the raw data, and one with the Procrustes transformed data.

The reported accuracies for a fitting are as following:
Accuracy for RFC on original data: 0.0286
Accuracy for RFC on Procrustes transformed data: 0.7429

2.3

In the above tasks, using Procrustes transformation, we perform a series of operations on the diatoms (point clouds) to align and standardise towards the target diatom. These operations affect the data points, and this removes some of the spatial information. The operations performed by Procrustes are translation, rotation and scaling, albeit we lose information about location, size and orientation.
One can argue in a case where the diatoms classes are directly correlated to their shape, the shape is what we look to preserve the most.
If the goal and practical appliance of such a classifier were to classify the various diatoms, it exactly proves beneficial to perform Procrustes transformation and loose the information described previously, as proven by the accuracy obtained above.

3 Transformations on images - Translation

1. Let x' and y' be the coordinates for $\tilde{I}$. In order to express $\tilde{I}$ in terms of x and y we observe that

$$x' = x + t_x = x + 1$$
$$y' = y + t_y = y + 0 = y$$

2. The desired matrix is given by

$$T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

   We can then use the above transformation to produce $\tilde{I}$ by the following

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = T \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

   This gives all the coordinates for $\tilde{I}$, which produces $\tilde{I}$.

3. It is possible to formulate this translation using a linear filter. The linear filter is given by $\begin{pmatrix} 0 & 1 \end{pmatrix}$. Notice that we only need 1 linear filter in the horizontal direction in order to get the desired translation.

4. The desired white square image with a black background can be seen in figure 7. The image has size 5×5.

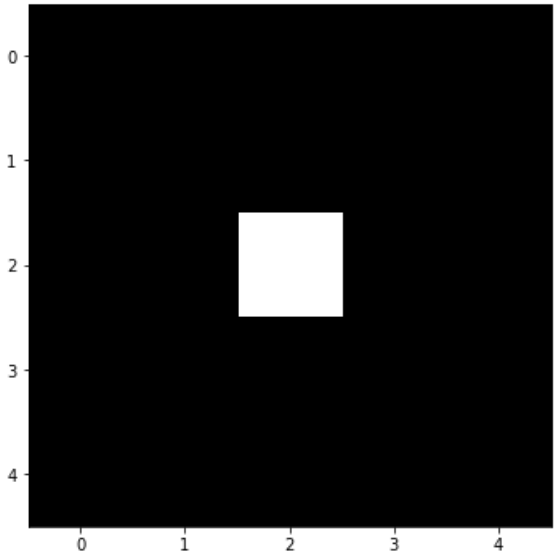

Figure 7: White square with black background (5x5).

5. The implementation is given by

```
def filter_translation(I,dx,dy):
    I = I.astype(np.float64)
    I_shift = fftconvolve(I,dx,mode = "same")
    I_shift = fftconvolve(I_shift,dy,mode = "same")
    return I_shift
```

There are many different boundary conditions possible. Some of the options are called constant, nearest and grid-wrap [2]. From Scipy we see that the constant boundary condition means that the input is extended by filling all values beyond the edge with the same constant value, defined by some parameter. The parameter could for example be equal to one. No interpolation is performed beyond the edges of the input. Moreover, the nearest boundary condition means that the input is extended by replicating the last pixel, and the grid-wrap boundary condition means that the input is extended by wrapping around to the opposite edge.

We applied this implementation in order to obtain two different translations. The results can be found in figure 8.

6. We will now implement a new function, which is able to handle non-integer translation, because of the use of nearest neighbor interpolation. We will use the homogeneous transform from Q3.2. The implementation is influenced by the websites, which discuss nearest neighbor interpolation and shift transformation [3] [4], and is given by

```
def space_translation(I,dx=0.6,dy=1.2):
    # https://theailearner.com/tag/cv2-warpaffine/
    # https://kwojcicki.github.io/blog/NEAREST-NEIGHBOUR

    rows,cols = I.shape
    # Transformation matrix
    M = np.array([[1,0,dx],[0,1,dy],[0,0,1]])
    orig_coord = np.indices((cols, rows)).reshape(2,-1)
    # Vertical stacking.
    orig_coord_f = np.vstack((orig_coord, np.ones(rows*cols)))
    print(orig_coord_f.shape)
    transform_coord = np.dot(M, orig_coord_f)
```

[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.shift.html
[3] https://kwojcicki.github.io/blog/NEAREST-NEIGHBOUR
[4] https://theailearner.com/tag/cv2-warpaffine/

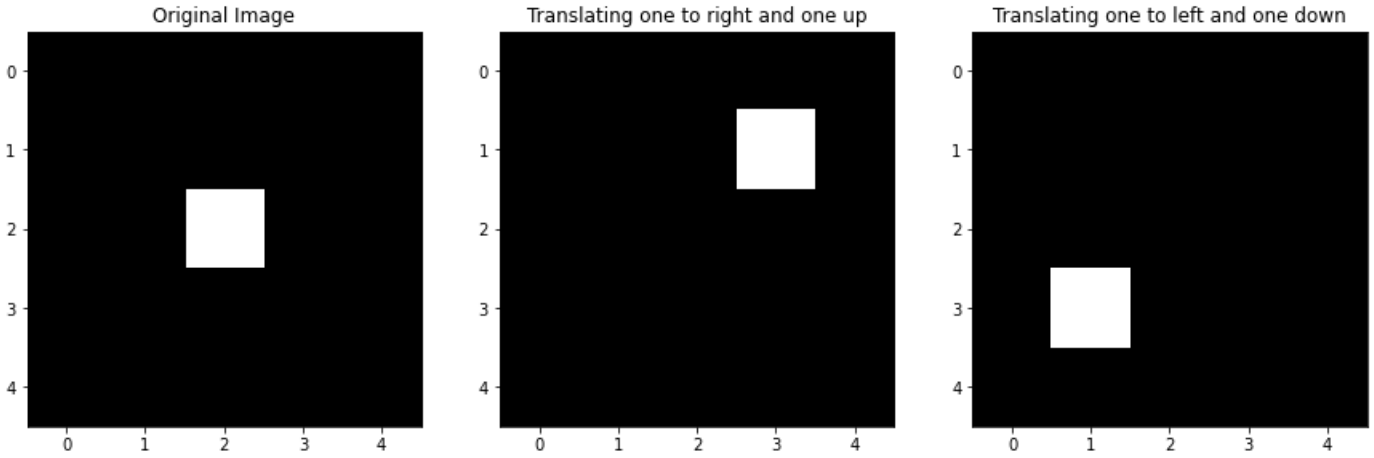

Figure 8: Integer translations using linear filters.

```
# Rounding towards zero (Nearest Neighbour Interpolation)
transform_coord = transform_coord.astype(np.int32)
# Keep only the coordinates that fall within the image boundary.
indices = np.all((transform_coord[1]<rows, transform_coord[0]<cols, transform_coord[1]>=0, transform_c
# Create a zeros image and project the points
I_new = np.zeros_like(I)
I_new[transform_coord[1][indices], transform_coord[0][indices]] = I[orig_coord[1][indices], orig_coord
return I_new
```

The illustration of the original white square image and the image you get by translating by $\mathbf{t} = \begin{pmatrix} 0.6 & 1.2 \end{pmatrix}^T$ can be found in figure 9. From figure 9 we see that this translation is equivalent to translating the image by $\mathbf{t} = \begin{pmatrix} 0 & 1 \end{pmatrix}^T$ due to nearest neighbor interpolation.

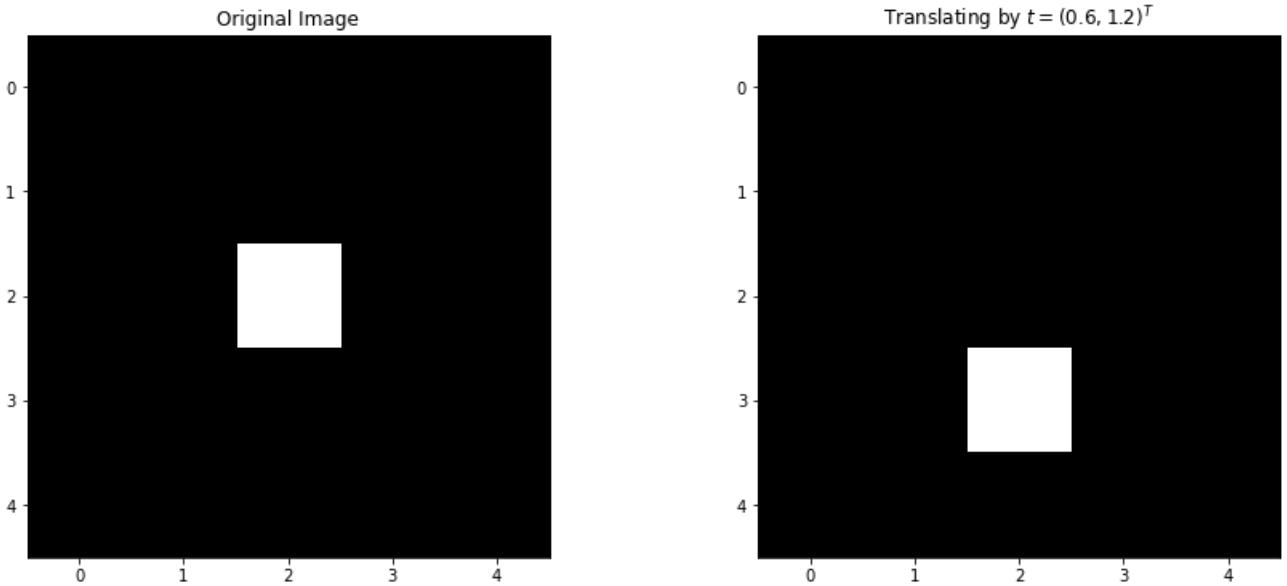

Figure 9: Non-integer translation.

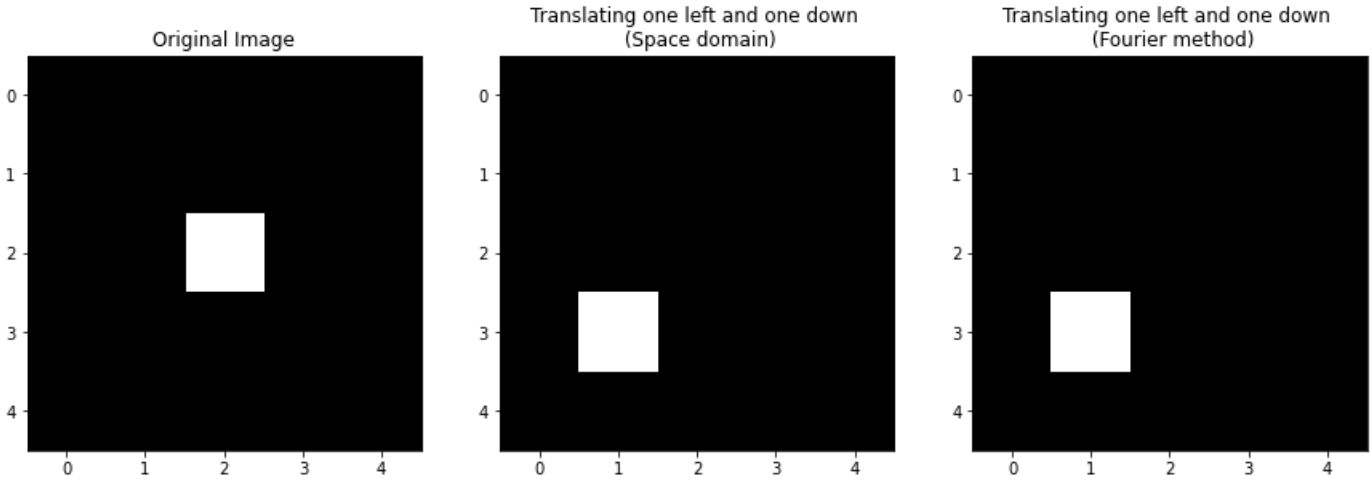

Figure 10: Comparing result images.

7. Using the equation of the Fourier transform of a translated image with respect to the Fourier transform of the original we get the following implementation:

```
def Fourier_translation(I,dx,dy):

    input_ = np.fft.fft2(I)
    freqs = np.fft.fftfreq(len(I))
    result = np.array([np.exp(-complex(0,1)*2*np.pi*f*dx) for f in freqs])*input_
    result = np.array([np.exp(-complex(0,1)*2*np.pi*f*dy) for f in freqs]).reshape(-1,1)*result
    result = np.fft.ifft2(result)
    return result.real
```

We compare the two result images in figure 10. From figure 10 we see that the two function produce the exact same image for integer translation.

8. it's not possible to do non-integer translations in the frequency domain. The index correction by the use of the nearest neighbor interpolation is not applicable in the frequency domain. This means that the shifting in the frequency domain for non-integer translation results in white square images having a gray transition between the black background and the white square. This creates gray squares. We have compared our implementation and the function *fourier_shift* [5] provided by Scipy in order to show that we get the same result image just described. We show the result images from non-integer translations in the frequency domain in figure 11.

[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.fourier_shift.html

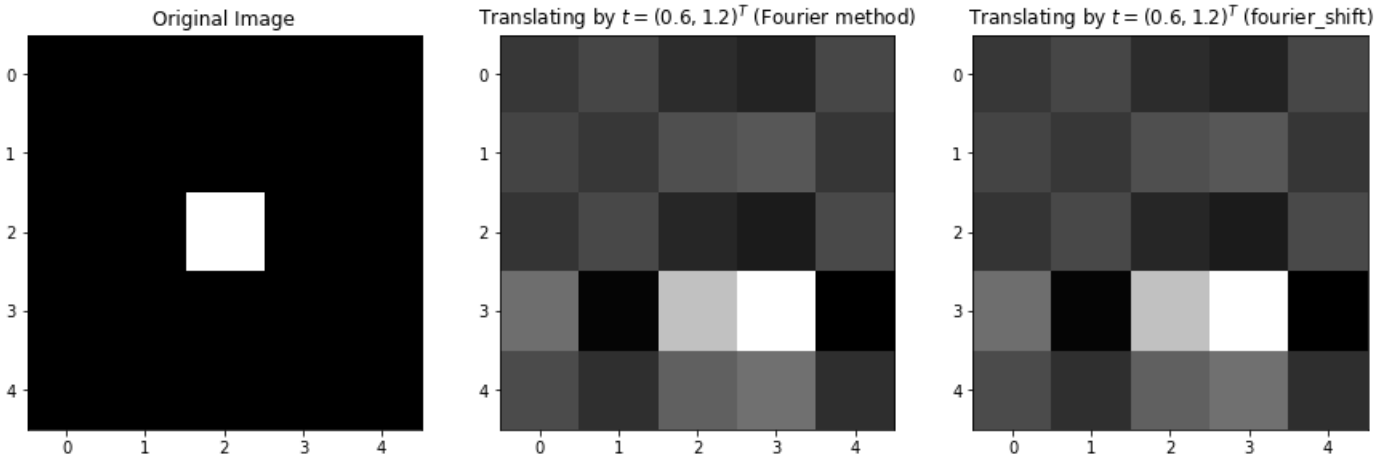

Figure 11: Applying the Fourier method and *fourier_shift* to the white square image.

The results from applying it also on the images *cameraman.tif* and *circles.png* can be found in figure 12. The gray effect described above is even more visible for those images.

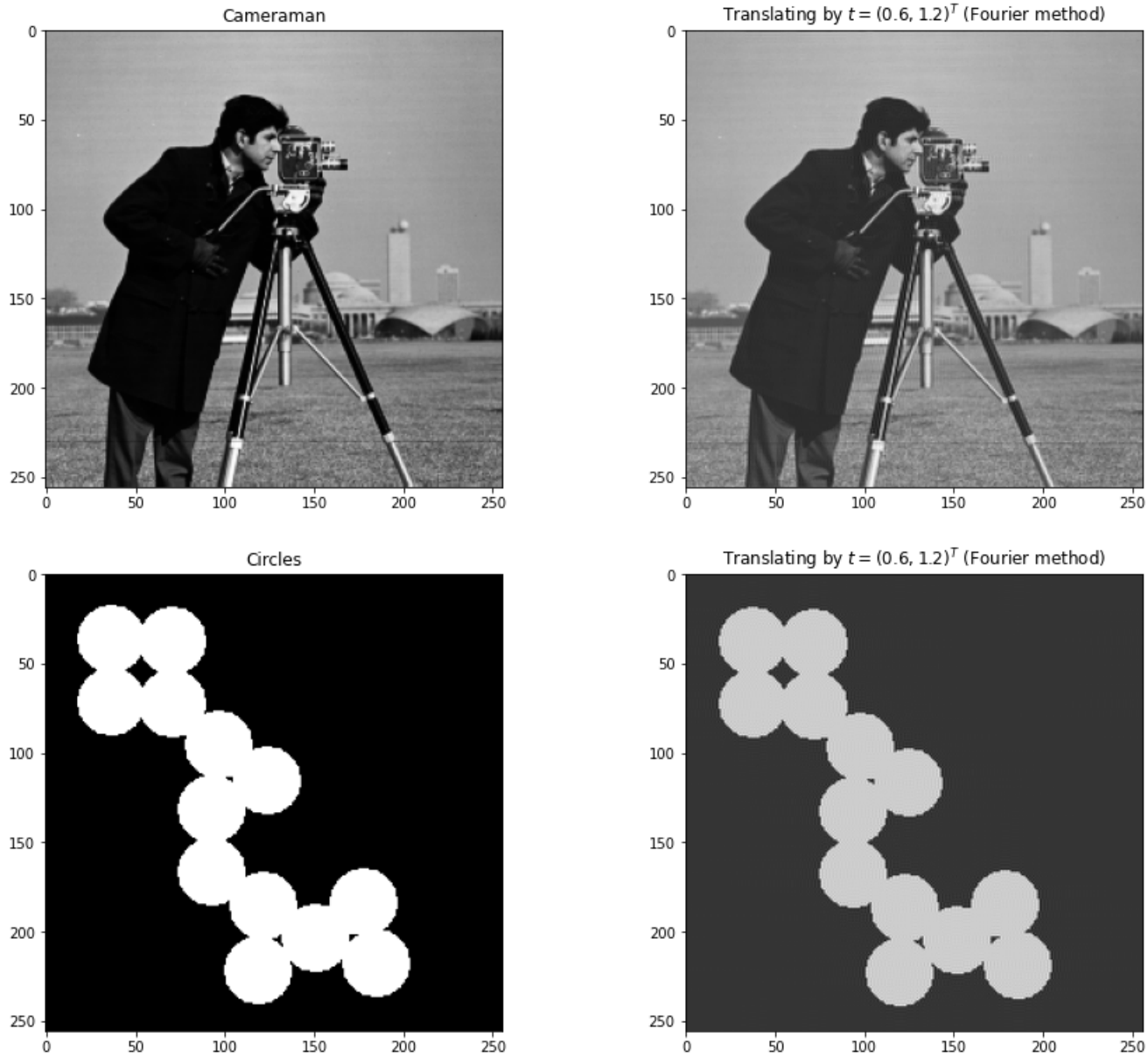

Figure 12: Applying the Fourier method to the images *cameraman.tif* and *circles.png*.

Appendix

Task 2.1

```
def task2_1(plot=False):
    X_train, X_test, X_train_y, X_test_y = setup()

    target_points = np.stack((X_train[0, ::2], X_train[0, 1::2]), axis=1)

    train_xs = X_train[0:, ::2]
    train_ys = X_train[0:, 1::2]

    # example usage mtx2s[diatom idx][:,0] all x values for post-procrustes for diatom idx in train_points:
    train_pro = procrustes_tranform(X_train)

    # Standardize target point to illustrate with post-procrustes example diatoms
    target_standard, _ = procrustes(target_points, target_points)

    if plot:
        fig, axs = plt.subplots(1, 3, figsize=(12, 4))

        axs[0].scatter(target_points[:,0], target_points[:,1], color='r', marker='D')
        axs[1].scatter(target_points[:,0], target_points[:,1], color='r', marker='D', alpha=.5)
        axs[2].scatter(target_standard[:,0], target_standard[:,1], color='r', marker='D', alpha=.25)
        axs[0].set_title("Target diatom")

        axs[1].scatter(train_xs[1], train_ys[1], color='b', marker='1')
        axs[1].scatter(train_xs[2], train_ys[2], color='g', marker='2')
        axs[1].scatter(train_xs[3], train_ys[3], color='y', marker='3')
        axs[1].set_title("Pre-Procrustes Diatoms Original")

        axs[2].scatter(train_pro[1][0::2], train_pro[1][1::2], color='b', marker='1')
        axs[2].scatter(train_pro[2][0::2], train_pro[2][1::2], color='g', marker='2')
        axs[2].scatter(train_pro[3][0::2], train_pro[3][1::2], color='y', marker='3')
        axs[2].set_title("Post-Procrustes Diatoms")

        fig.suptitle("Procrustes Transformation with target diatom in red")
        fig.tight_layout()
```