Image Processing
Lab 4

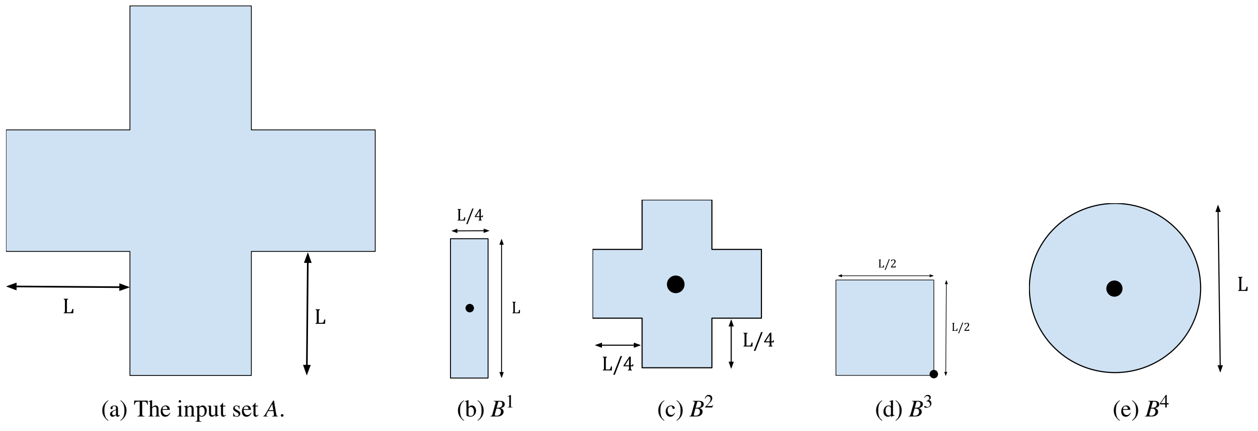
Kevin Gevers (s25595987)
Jeroen Overschie (s2995697)
Group 01

January 20, 2021

Note, that all used source code can be found attached next to this report's pdf file. Its structure should be self-explanatory; all requested functions are named accordingly and any extra functions are explained in the report. Note that (almost) every function has a corresponding test script, which is named just like its function, but with a suffix '`_test`'. Where possible, we followed the terminology from the book [1] for variable naming.

Exercise 1

In this first exercise, we are asked to solve Problem 9.9 from the book, from the Morphological Image Processing chapter. The exercise entails sketching dilations and erosions from some (binary) set using various Structuring Elements (SE's). Let us first illustrate the set A on which the morphological operations will be applied, and the respective Structuring Elements, B^1, B^2, B^3, and B^4.



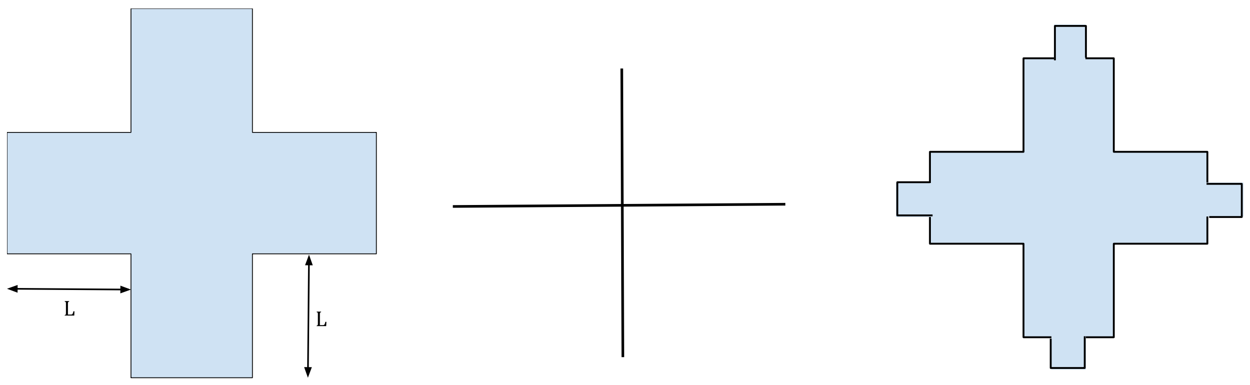
(a) The input set A. (b) B^1 (c) B^2 (d) B^3 (e) B^4

Figure 1: An overview of the Structuring Elements $B^1 ... B^4$ to be used in Exercise 1, along with its input set, A.

A notable observation is that, for the Structuring Elements, it is not always the case that the origin lies in the middle of the shape. For most, the origin is indeed in the shape center, but for B^3 (see Figure 1d) the origin is at the bottom-right point of the shape. This is important to note since it has an effect on the dilation/erosion procedures. For the other shapes, the origin lies at the center point.

(a)

In the first assignment, we first erode A by B^4 and then dilate the result by B^2, i.e. $(A \ominus B^4) \oplus B^2$. The result of the operation can be seen in Figure 2.



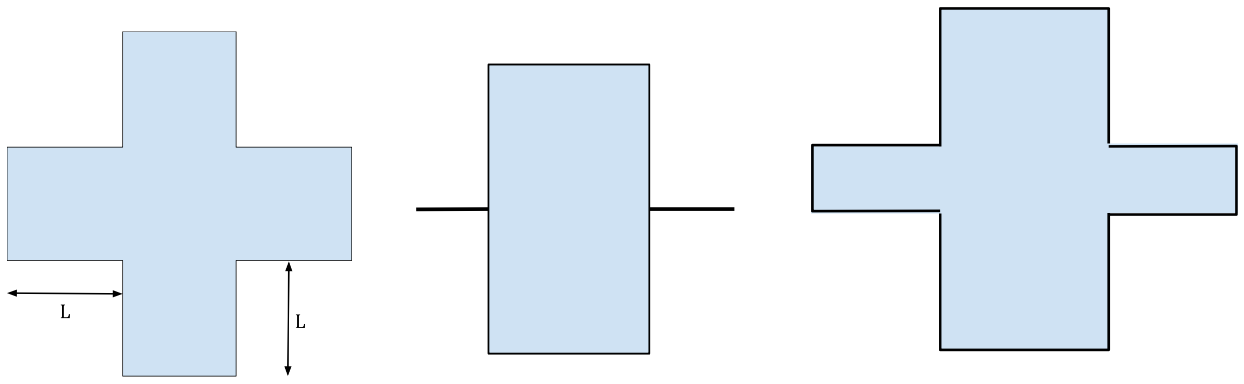
(a) The input set A. (b) $(A \ominus B^4)$ (c) $(A \ominus B^4) \oplus B^2$

Figure 2: Overview of the erosion and dilation of assignment (a), using the SE's B^4 (Figure 1e) and B^2 (Figure 1c).

It can be observed that the first erosion strips off almost all foreground pixels. The circle shape only fits a very select amount of pixels, leaving out only what looks like the 'skeleton' of the original cross shape. In fact, since the circle is of diameter L, meaning it can 'perfectly' fit the sections of the cross, it can be seen as a *maximum disk* for A. In the following dilation operation, many foreground pixels are added around the cross shape, and including some interesting looking 'tips' at the end of the cross points.

(b)

Next, we erode by B^1 and then dilate by B^3, i.e. $(A \ominus B^1) \oplus B^3$. The result of this operation can be seen in Figure 3.



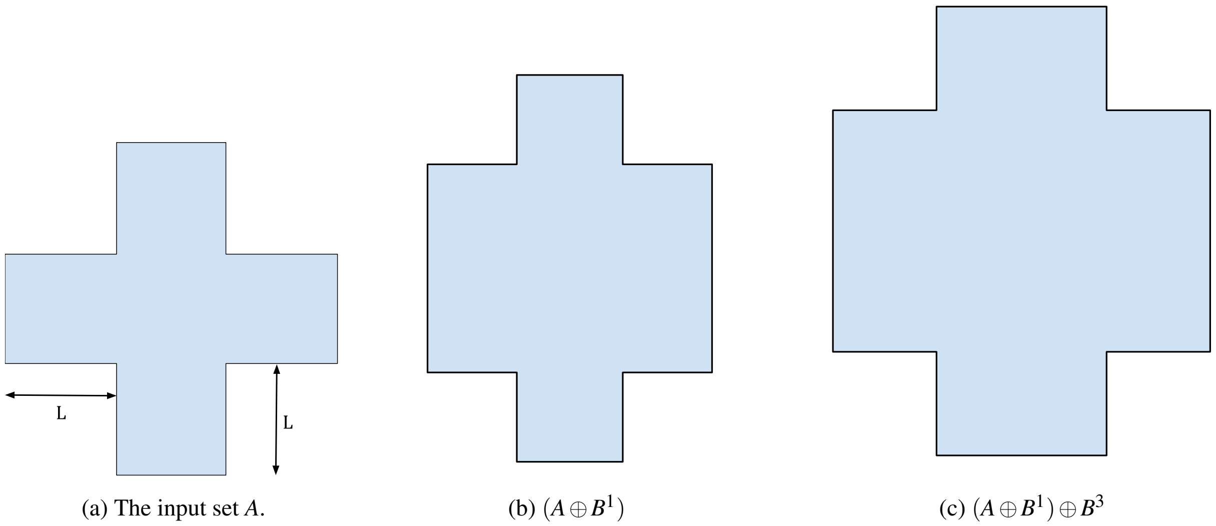
(a) The input set A. (b) $(A \ominus B^1)$ (c) $(A \ominus B^1) \oplus B^3$

Figure 3: Overview of the erosion and dilation of assignment (b), using the SE's B^1 (Figure 1b) and B^3 (Figure 1d).

It can be seen that the erosion of the shape by the rectangular SE makes that much of the vertical foreground pixels remain, but where the rectangular shape does fit is mainly in the horizontal cross spacing. Because the height of the rectangular SE B^1 (Figure 1b) is exactly L, it fits the left- and right sections of the cross perfectly. Because the origin is at the center, what is left after the erosion operation is only a small horizontal section. The dilation operation, on the other hand, then again increases the number of foreground pixels by enlarging the shape. The fact that the B^3 origin is at the bottom right position of the shape does not much influence the dilation operation (it would for the erosion operation).

(c)

Lastly, we dilate by B^1 and then dilate again, but by B^3, i.e. $(A \oplus B^1) \oplus B^3$. The result of this operation can be seen in Figure 4.



(a) The input set A. (b) $(A \oplus B^1)$ (c) $(A \oplus B^1) \oplus B^3$

Figure 4: Overview of the two dilation operations of assignment (c), using the SE's B^1 (Figure 1b) and B^3 (Figure 1d).

Both dilation operations enlarge the shape. The first, however, does so mainly in the vertical estate; using the B^1 SE the most hits occur in the vertical direction. We do get some horizontal enlargement too, but this is confined to a little less than $L/4$ on all horizontal edges. The last dilation operation enlarges the entire shape in all directions. Note again that the non-centered origin of B^3 does not have a major impact on the result of the dilation operation.

Exercise 2

(a)

In this first assignment, we are asked to implement morphological **dilation**. The goal is to perform **binary** dilation using some arbitrary *Structuring Element* (SE) of size 3×3. Because we are performing binary dilation, we have to assume and make sure our input images are also binary, i.e. have only 1's and 0's in them - which can be neatly described in Matlab using the `logical` type. Furthermore, we assume that the origin of the SE lies in the center. That said, let us first describe the applicable theory on Morphology in image processing.

The effect of a morphological dilation operation can be interpreted as "growing" or "thickening" objects in an image. Note that with objects we mean the *foreground* pixels of an image (1's) - whilst the background is represented by *background* pixels (0's). Because a dilation thickens image objects given a suitable SE, it can be used to bridge small gaps between otherwise continuous image components. It is even more useful when performed after an erosion, which is a morphological *opening* - which is a technique that can be used to remove small noise, smoothen object contours and break narrow bridges.

Since dilation is best defined using notation and terminology from **set theory**, we need some prior knowledge first. We first denote A as being a set of foreground pixels, such that $A \subseteq I$, i.e. it is part of the input image I. Also, since we are using set theory, we need to define our image in suitable notation too. To define a binary image, we can conveniently decide to regard **foreground pixels only**, such that we can define the entire image in one set of 2-dimensional tuples representing the (x, y) coordinates. Each tuple can be said to be part of the 2-D integer space, i.e. Z^2. To obtain the background pixels, we can take the complement of the foreground pixel set, i.e. A^c. Note that even though the complement means **all** objects not in A (i.e. $A^c = E \setminus A$; the difference with the universal set), in the case of working with a digital image our dimensions are actually bounded by the image border - meaning that the set of background pixels can also be expressed as a discrete set.

Given that A and B are sets in Z^2, we can now define dilation as:

$$A \oplus B = \{z | (\hat{B})_z \cap A \neq \emptyset\}, \tag{1}$$

in which $\hat{B}$ denotes the *reflection* of a set around its origin:

$$\hat{B} = \{w | w = -b, \text{ for } \in B\}, \tag{2}$$

and $(B)_z$ denotes the translation of the set B by point $z = (z_1, z_2)$:

$$(B)_z = \{c | c = b + z, \text{ for } b \in B\} \tag{3}$$

This intuitively means that we reflect B by its origin and keep all translations z where the foreground elements of $\hat{B}$ overlap with at least one element of A. When we have obtained the set of all translations z that obey these conditions, it means that the dilation of A by B is equal to that set of translations: remember we consider only the foreground pixels in our representation of the binary image, so the dilation output matches up with our data representation.

To implement the dilation operation in Matlab, we recognized both the dilation and erosion operations had similar functionality that could be shared - under the principle of writing DRY (*Don't Repeat Yourself*) code. Therefore we first wrote a general-purpose function for both operations, IPMORPH, which can be found in ❖ Listing 1. This function loops the input image pixels one-by-one and then 'overlaps' the SE with the pixel neighborhood centered at the origin, which is at the SE center point. To better facilitate overlapping the SE around the entire input image, padding is created around the input image. In the case of erosion, the padding consists out of 1's - whilst for dilation, these are 0's.

Note that we do not use a strictly set-theoretical approach in our Matlab algorithm - this was done with the computational speed in mind. If we adhere strictly to the set-theoretical approach as described in the book, we lose lots of computational speed. Specifically, we tried using the INTERSECT Matlab function to perform the intersection ($\cap$) from Eq 1, however, the function remained too slow: executing just one dilation would take many seconds. The same applies to erosion: as we will see later in computing the erosion there also exists the possibility of using set inclusion ($\subseteq$) - which could be implemented using the ISMEMBER Matlab function. This, however, turned out even slower than INTERSECT.

That said, the dilation function IPDILATE can be found in ❖ Listing 2, which uses the IPMORPH function like described above. More specifically, in the case of dilation the Matlab ANY function is used to implement the SE 'hitting' any foreground pixels within its neighborhood, like described in Eq 1. From the neighborhood, we first seek out only the applicable pixels by selecting only those in the SE by using A(B). Also, $\hat{B}$ is computed using ROT90 and a transpose ($\acute{}$).

(b)

Next up is morphological **erosion**. Erosion shrinks or thins objects in a binary image. Using the notation like in (a), we can denote the erosion as follows:

$$A \ominus B = \{z | (B)_z \subseteq A\} \tag{4}$$

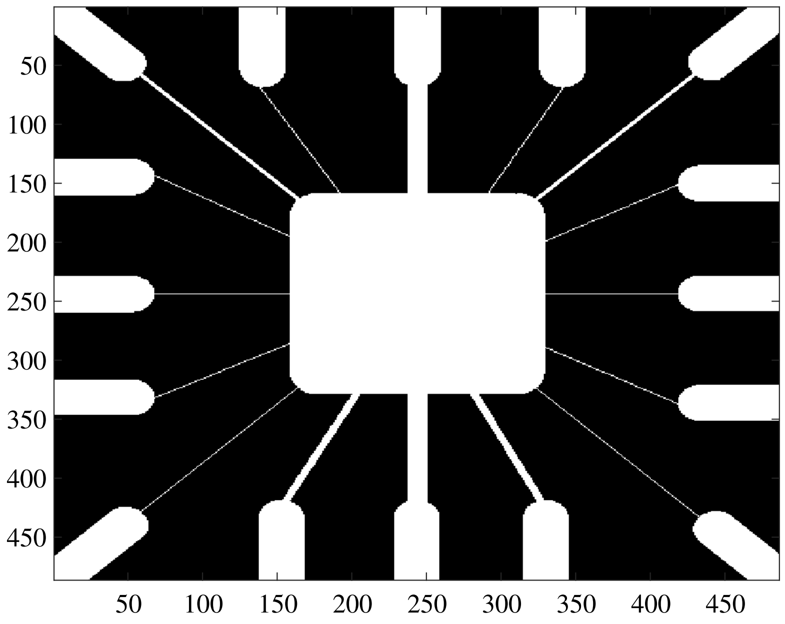
which can also be written using an intersection operation onto the object complement A^c:

$$A \ominus B = \{z | (B)_z \cap A^c = \emptyset\} \tag{5}$$

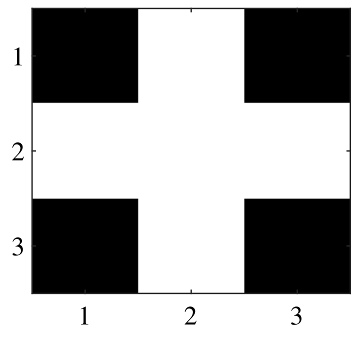
Which concludes the mathematical notation of the erosion operation. Intuitively it can be understood that erosion needs the SE to exactly 'fit' all pixels in its neighborhood; only then will an object pixel 'remain'. The erosion operation was implemented in IPERODE and can be found in ❖ Listing 3. To implement the SE 'fitting' the foreground pixels, the ALL Matlab operation is used.

(c)

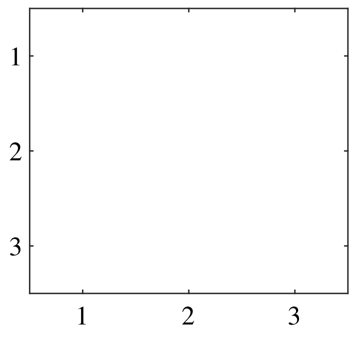
Finally, we test both our functions by performing the operations on the `wirebondmask.tif` image and using various SE's. The script for testing can be find in ❖ Listing 4. The image is of size (486 × 486) and is already in the binary format we need, i.e. has the Matlab `logical` type. We devised three SE's to test our functions: a 3×3 cross, a 3×3 square and one with higher dimensions, a 15×15 but with its top-left corner set as 0's. The latter SE is to make sure our algorithm is compatible with non-symmetrical SE's and still correctly computes the dilation using its reflection. For an overview of the input image and SE's used, see Figure 5.



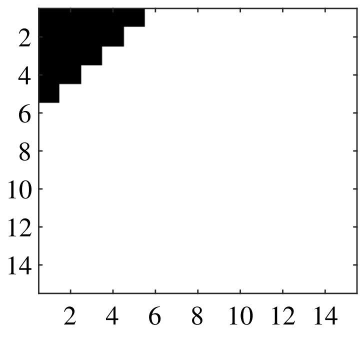
(a) The input image *I*.



(b) *Bcross* (3x3 SE)



(c) *Bsquare* (3x3 SE)



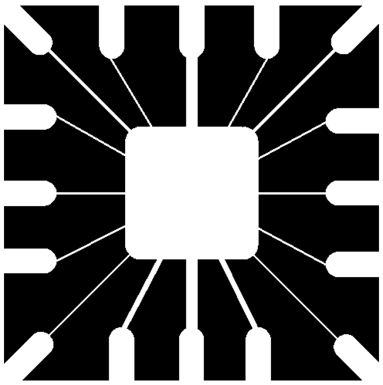
(d) *Bbigsquare* (15x15 SE)

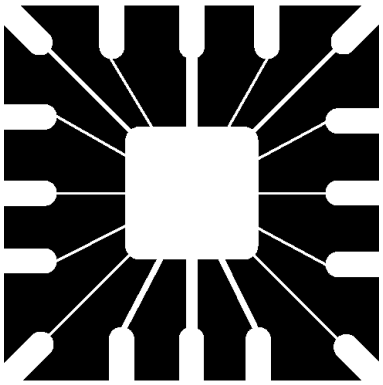
Figure 5: Overview of the input image *I* and the Structuring Elements (SE's), *Bcross*, *Bsquare* and *Bbigsquare* used to test the morphological functions that follow. Two 3×3 symmetrical SE's are defined and one non-symmetrical 15×15 SE to investigate functionality using a larger and non-symmetric SE.

Let us now first investigate running the dilation operation on the `wirebondmask.tif` image.

Dilation

Dilation was tested on the three different SE's, which can be seen in Figure 6. It can be observed that given an SE with more foreground pixels or of a larger size, the image is dilated to increasingly larger sizes, i.e. more foreground pixels are added to the resulting image.


(a) $I \oplus Bcross$


(b) $I \oplus Bsquare$

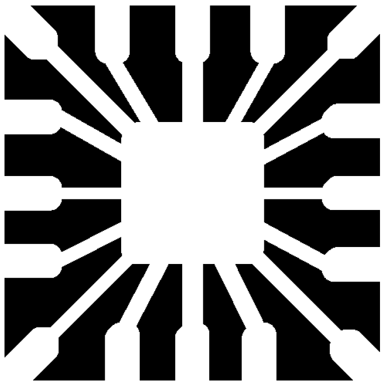
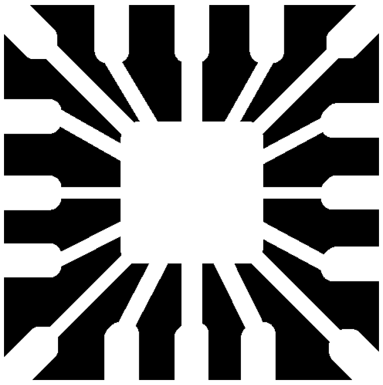
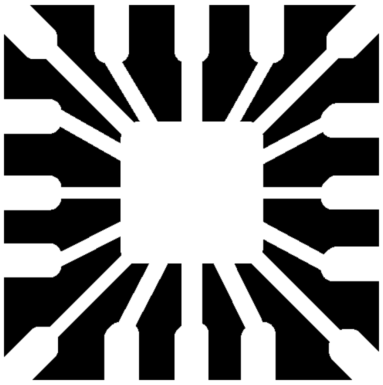

(c) $I \oplus Bbigsquare$

Figure 6: Dilation transformations on *I* (Figure 5a) using the IPDILATE function and the Structuring Elements like defined in Figure 5.

Next, we also wanted to confirm our dilation function was operating like it should, by comparing it against a built-in Matlab function, IMDILATE. This is defined in the morphological operations section in Matlab, as part of the image processing toolbox. It can be conveniently used to confirm the correct functionality of our own function. We used the *Bbigsquare* SE and dilated by both our and Matlab's function and also included a difference-image to see the two differ at all. The result can be seen in Figure 7.


(a) IPDILATE


(b) IMDILATE

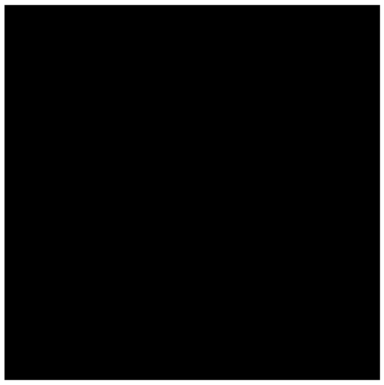
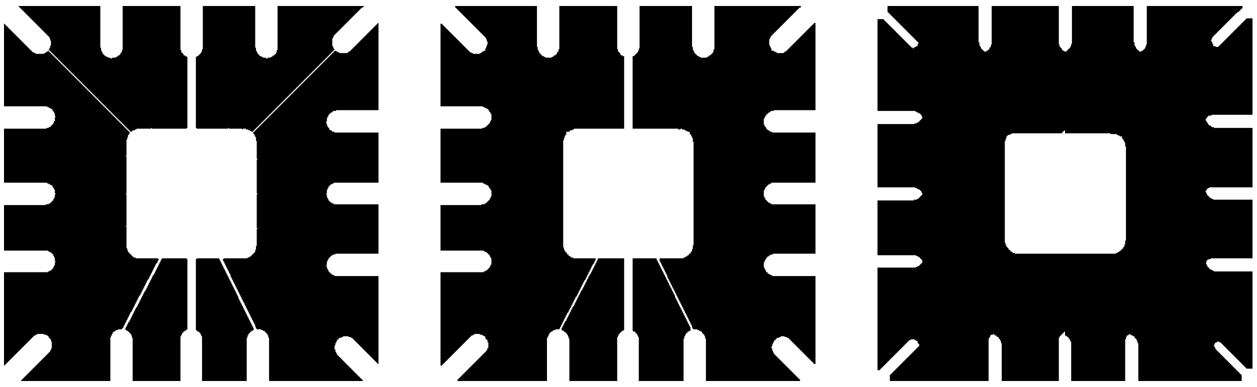

(c) Difference

Figure 7: $I \oplus Bbigsquare$ executed using both IPDILATE and IMDILATE, and the difference between the two results.

In Figure 7 it can easily be observed that the two functions return an exactly identical result, i.e. our function operates as it should.

Erosion

Next, we test our erosion function. Again, we perform erosion on our three SE's and the `wirebondmask.tif` image. See Figure 8.

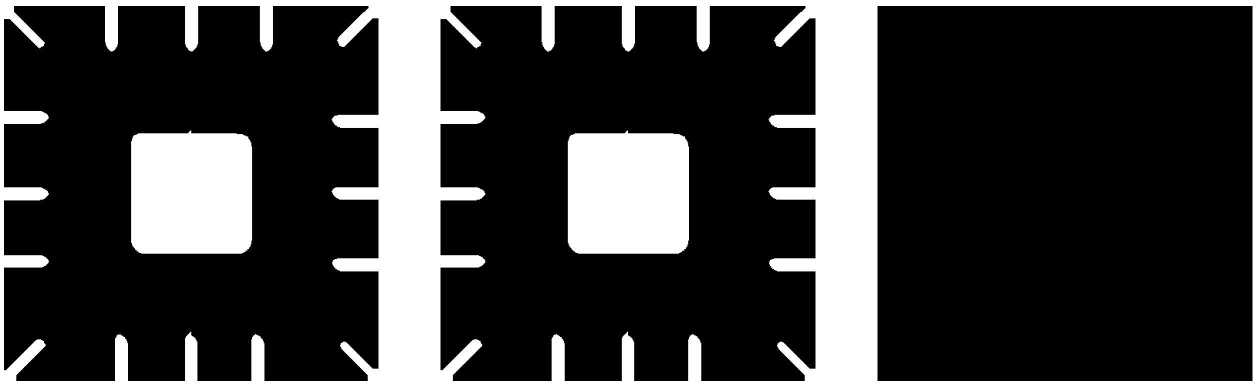


(a) $I \ominus Bcross$ (b) $I \ominus Bsquare$ (c) $I \ominus Bbigsquare$

Figure 8: Erosion transformations on I (Figure 5a) using the IPERODE function and the Structuring Elements like defined in Figure 5.

In Figure 8 it can be observed that the erosion removes some of the diagonal lines that can be seen in the image. And it does so more using larger SE's, which is like expected: at some point, the SE won't "fit" in the lines anymore, therefore removing the line.

Also for erosion, we create a comparison with a built-in Matlab function: this time IMERODE. The result can be seen in Figure 9.



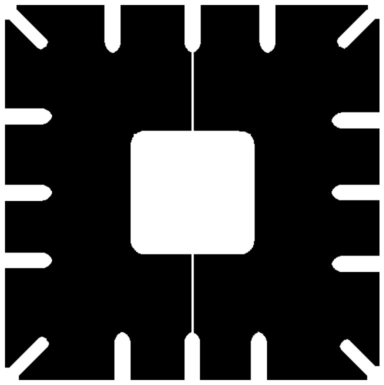
(a) IPERODE (b) IMERODE (c) Difference

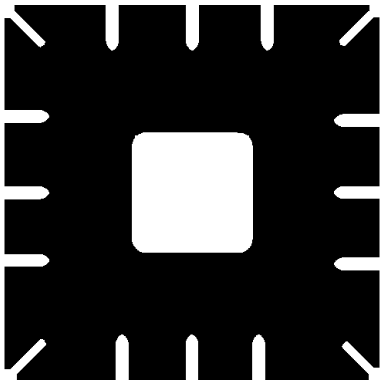
Figure 9: $I \ominus Bbigsquare$ executed using both IPERODE and IMERODE, and the difference between the two results.

Like before, Figure 9 shows that there is zero difference between our function and the built-in Matlab function, indicating the correctness of our function.

Recreating Figure 9.5 from the book

Finally, to conclude the testing of our functions, we thought it'd be interesting to re-create Figure 9.5 from the book [1]. We indeed managed to do this, see Figure 10.


(a) Erosion using 11×11 SE


(b) Erosion using 15×15 SE

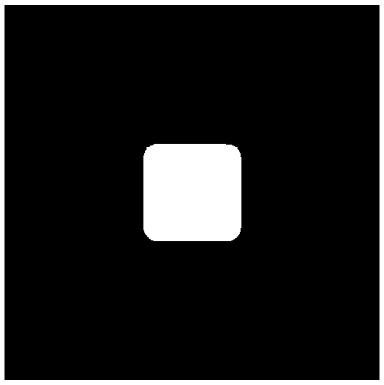

(c) Erosion using 45×45 SE

Figure 10: Erosion of the Wirebondmask image (Figure 5a), using various-sized square SE's, recreating part of Figure 9.5 from the book [1].

The plot (Figure 10) shows, that given square SE's of increasing size, the input image is increasingly eroded until only the center-piece remains. It can be observed that the result is indeed equal to the example in the book, which can be found in Chapter 9, Figure 9.5.

Exercise 3

In this exercise, we are tasked with creating an implementation of the morphological skeleton decomposition and reconstruction. Skeleton decomposition can be used as a controlled erosion process that allows for quick and accurate processing of images that would otherwise require computationally and memory expensive operations.

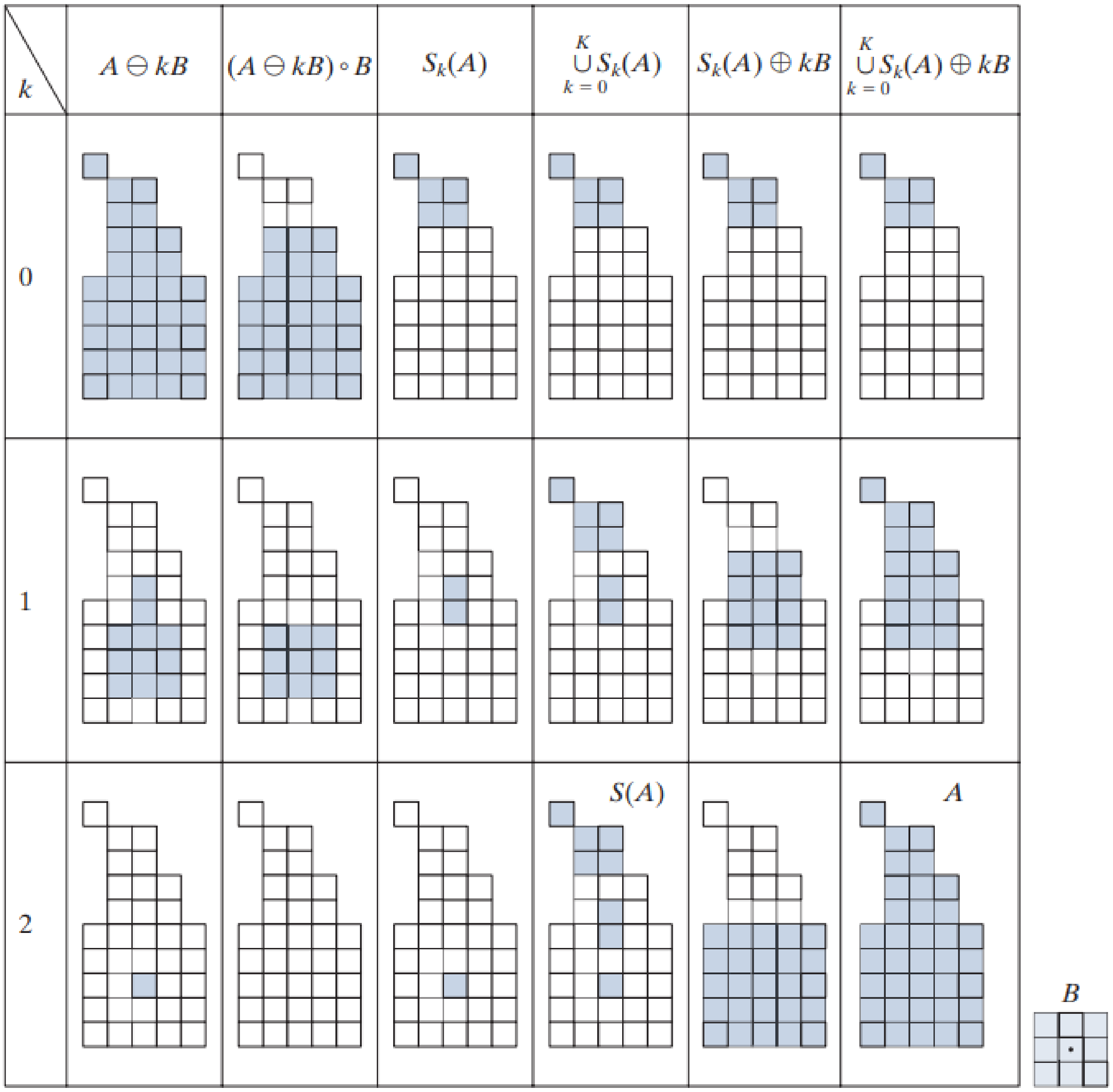


Figure 11: Image demonstrating the steps for the skeleton decomposition and reconstruction as presented in the book (fig. 9.26) [1]

(a)

The first part of the exercise is the skeleton decomposition. The function will require a binary input image and a structuring element (SE). The structuring element will be a 3 by 3 matrix for the purpose of this exercise. We will also assume (like in the previous exercise) that the structuring element contains the origin. The decomposition follows 4 steps that have to be done K times each. K is the determined by how many times erosion can be applied to the input image before the image is empty (Eq 6 applies the erosion). The process can be described in four (1-4) steps:

$$A \ominus kB \tag{6}$$

$$(A \ominus kB) \circ B \tag{7}$$

$$S_k(A) = (A \ominus kB) - ((A \ominus kB) \circ B) \tag{8}$$

$$\bigcup_{k=0}^{K} S_k(A) \tag{9}$$

Where: A = input image, B = structuring element, and S_k = skeleton for iteration k.

The first 4 columns of Figure 11 demonstrate how these four steps work for a simple binary image. The first step is applying erosion to the input image *k* times, which follows Equation 6. In the example from the book the image would be completely empty if erosion was applied 3 times, so $K = 2$. The second step is an opening applied to the result of step 1, which follows Equation 7. Step 3 is the skeleton for a specific *k*. This is done by subtracting the result of step 2 from the result of step 1. It follows Equation 8, which is Eq. 9-29 in the book. The fourth and final step of the decomposition is taking the union of the different skeletons S_k. Important to note is that for Equation 6 the erosion is applied *k* times, each time taking the result of the previous iteration. This can also be written as Equation 10, which is Eq. 9-30 in the book. Using all these steps the final skeleton is generated by taking the result of the highest *k* for step 4.

$$A \ominus kB = ((\ldots((A \ominus B) \ominus B) \ominus \ldots) \ominus B) \tag{10}$$

The assignment requires us to implement the skeleton decomposition using a recursive function. So each level of *k* should be done by recursion and not a loop. We implemented it as such by adding the result of the current level (S_k) and the result of the next level together, by calling the same function again. For this, we use the base case where applying erosion to the input image would result in an empty image. If this is the case the result of the current level is returned without calling the function again.

Because the result of the function would simply give the final skeleton and the different levels of the skeleton (S_k) could not be retrieved anymore, we need to encode the resulting images in such a way they can be retrieved. This needs to be done according to Equation 11, where we encode all the skeleton sets $S_k(A)$ by introducing a skeleton function *skf(A)*. This basically boils down to the skeleton of each level being added to the result with a different number, namely (*k+1*). This allows us to understand the different levels of the skeleton in the reconstruction step.

$$[\text{skf}(A)](i,j) = \begin{cases} k+1, & (i,j) \in S_k(A) \\ 0, & (i,j) \notin S_k(A) \end{cases} \tag{11}$$

The results of this function are presented in part (c) of the exercise and in Figure 12. The implementation of the function can be found in ❖ Listing 5.

(b)

The next part will be the reconstruction of the skeleton we created in part (a). To reconstruct the image two steps have to be executed. Each step has to be executed *k* times.

$$S_k(A) \oplus kB \tag{12}$$

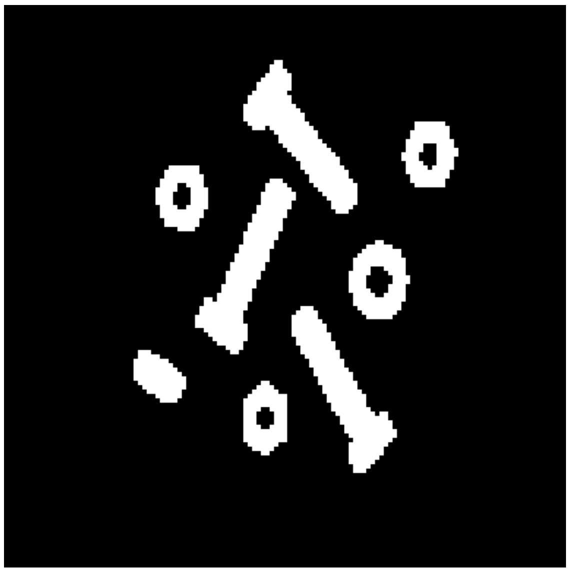
$$\bigcup_{k=0}^{K} S_k(A) \oplus kB \tag{13}$$

The first step, Equation 12, is the dilation of the skeleton for that level. The second step is the union of the results from the first step and follows Equation 13 (Eq. 9-32 in the book). The different levels of the skeleton can be distinguished as such due to the number written. The assignment requires the reconstruction to also make use of recursion instead of loops. For our implementation, we extract the pixel coordinates that contain the value 1 which is the result of the first step for each k. Afterward, we decrease all non-zero values in the remaining skeleton and apply the dilation for each number. In case certain coordinates would have multiple numbers due to the different dilations overlapping the highest number is kept. This means that the intermediate result is not correct in these cases, but the final result is correct. In the fifth column of Figure 11 the entry for $k = 1$ would not have the bottom 3 pixels, because those would be claimed by the ongoing erosion of $k = 2$. These intermediate levels can be correctly extracted, which we show on lines 33 to 35 in ❖ Listing 6, but are not passed to the next iteration of the recursion. We use recursion in the same way as we did for the skeleton decomposition, so we add the result of the current step and the result of a call to the same function together. The base case this time is an empty skeleton set, which means that all the skeleton sets have been processed.

The results of this function are presented in part (c) of the exercise and in Figure 13. The implementation of the function can be found in ❖ Listing 6.

(c)

The last part of the assignment requires us to show the results of our implementation of the skeleton decomposition and reconstruction using the input image `nutsbolts.tif`. We can first of all show both the input image and the skeleton decomposition. See Figure 12.


(a) Input image, `nutsbolts.tif`

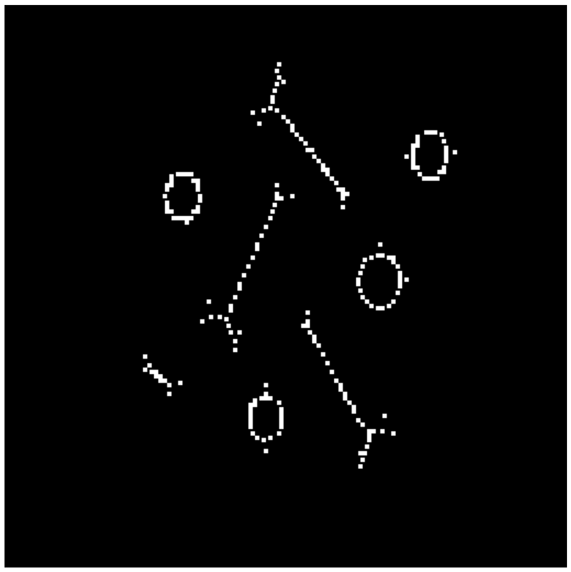
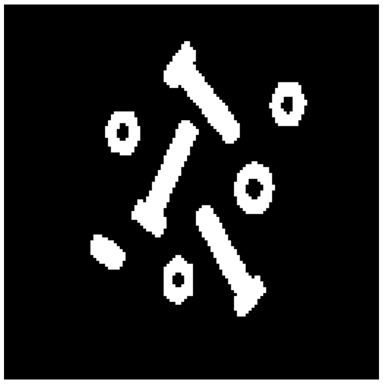
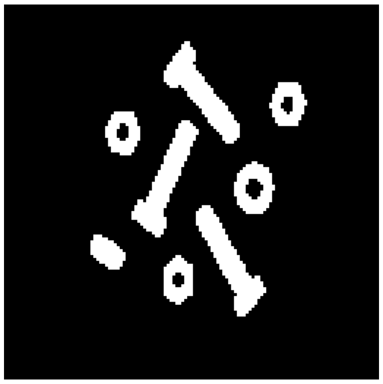

(b) The complete skeleton image. This is produced by passing the binary input image 12a to the IPSKELETONDECOMP function.

Figure 12: The results from applying our implementation of skeleton decomposition and reconstruction to the input image `nutsbolts.tif`.

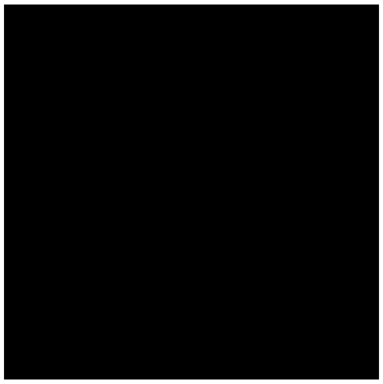
In the plot (Figure 12), the original image can be seen on the left-hand side, whilst the decomposition is shown on the right side. Indeed the pixels are placed at locations that make sense for the skeleton - pixels which, if they were used for dilated in the right way, could lead to a reconstruction of the input image. Let's see whether worked out using the IPSKELETONRECON function. See Figure 13.



(a) Input image, `nutsbolts.tif`



(b) The reconstruction of the image. This is the produced by passing the result of IPSKELETONDECOMP to the IPSKELETONRECON function.



(c) The difference image of the input image and the reconstructed image. The absolute difference is 0 here (this is verified in the code, see ❖ Listing 7), meaning the images are exactly the same and this image is completely black.

Figure 13: The results from applying our implementation of skeleton decomposition and reconstruction to the input image `nutsbolts.tif`.

In the plot (Figure 13) the input image, the reconstructed image and the difference between the input image and reconstructed image are shown. The difference image is completely back, meaning that the reconstruction is a perfect match for the input image. This is also verified by taking the absolute difference between the two images in the code.

To demonstrate our implementations we created a test file that generates the skeleton and reconstruction images, this can be found in ❖ Listing 7.

Individual contributions

- **Exercise 1 and 2**. Contribution to program design, program implementation, answering questions posed and writing the report: *Jeroen* 100%.
- **Exercise 3**. Contribution to program design, program implementation, answering questions posed and writing the report: *Kevin* 100%.

References

[1] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J., 2008.

A Code

A.1 Exercise 2

Listing 1: IPmorph.m: perform morphological transformations, namely *dilation* and *erosion*, using a single function.

```
function Imorph = IPmorph(I, B, type)
% IPmorph Applies a morphological transformation to a binary image using
%         some structuring element. It is assumed the SE origin is at its
%         center.
%   Arguments:
%       I: input image to transform. Must be binary, i.e. have logical values.
%       B: structuring element (SE). Must have odd dimensions and also be
%       binary, i.e. have logical values. Its origin is automatically set
%       at its centerpoint.
%       type: morphological transformation type. Can be either 'erode' or
%       'dilate'.
%   Returns: morphologically transformed image

% Dimensions
[M, N] = size(I);       % height, width
[Mse, Nse] = size(B);   % height, width
% Ensure image and SE is binary and of logical type
assert(islogical(I));
assert(islogical(B));
% Ensure SE has odd dimensions
assert(rem(Mse, 2) ~= 0)
assert(rem(Nse, 2) ~= 0)
% Ensure type argument is of acceptable value
assert(ismember(type, {'erode', 'dilate'}));

% Put padding around original image. Then, SE can freely move around.
pad_y = floor(Mse / 2);
pad_x = floor(Nse / 2);
switch type
    case 'erode'    % pad with 1's in case of erosion
        Ipad = padarray(I, [pad_y, pad_x], 1, 'both');
    case 'dilate'   % pad with 0's in case of dilation
        Ipad = padarray(I, [pad_y, pad_x], 0, 'both');
end

% Loop image pixels
Imorph = false(M, N);
for y=(1:M)+pad_y
    for x=(1:N)+pad_x
        SE_ycoords = (y - pad_y):(y + pad_y);
        SE_xcoords = (x - pad_x):(x + pad_x);
        A = Ipad(SE_ycoords, SE_xcoords);

        switch type
            case 'erode'  % erode:  1 iff SE 'fits' neighborhood exactly.
                value = all(A(B));
            case 'dilate' % dilate: 1 iff SE 'hits' any in neighboorhood.
                Bhat = rot90(B, 2)';
                value = any(A(Bhat));
        end
        Imorph(y - pad_y, x - pad_x) = value;
    end
end
end
```

A.1.1 Exercise 2 (a)

Listing 2: IPdilate.m: perform dilation using IPMORPH (Listing 1).

```
function Idil = IPdilate(I, B)
% IPdilate Dilates a binary image using some structuring element.
%   Arguments:
%       I: input image to dilate. Must be binary, i.e. have logical values.
%       B: structuring element (SE). Must have odd dimensions and also be
%       binary, i.e. have logical values. Its origin is automatically set
%       at its centerpoint.
%   Returns: dilated image
Idil = IPmorph(I, B, 'dilate');
end
```

A.1.2 Exercise 2 (b)

Listing 3: IPerode.m: perform dilation using IPMORPH (Listing 1).

```
function Iero = IPerode(I, B)
% IPerode Erodes a binary image using some structuring element.
%   Arguments:
%       I: input image to erode. Must be binary, i.e. have logical values.
%       B: structuring element (SE). Must have odd dimensions and also be
%       binary, i.e. have logical values. Its origin is automatically set
%       at its centerpoint.
%   Returns: eroded image
Iero = IPmorph(I, B, 'erode');
end
```

A.1.3 Exercise 2 (c)

Listing 4: IPdilate_erode_test.m: Test script to demonstrate the functionality of both IPDILATE and IPERODE. Saves images to .svg files.

```
clc;                                    % clear the command window
close all;                              % close open figure windows

imname = 'wirebondmask';
inputfile = ['input_images/', imname, '.tif'];
f = imread(inputfile);                    % read input image

% Original image
figure('visible', 'off');
colormap(gray(256));
imagesc(f);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Write current figure to file
all_file = ['output_plots/', imname,'_original','.svg'];
saveas(gcf, all_file);
```

```
fprintf('\nComplete image has been saved in file %s\n', all_file);

% Show Morphological operations per Structuring Element
%% Cross
Bcross = logical([0 1 0; 1 1 1; 0 1 0]);
figure('visible', 'off');
colormap(gray(256));
imagesc(Bcross);
xticks(unique(round(get(gca, 'xTick')))); % only whole value tick labels
yticks(unique(round(get(gca, 'yTick')))); % only whole value tick labels
set(gcf, 'PaperUnits', 'normalized')
set(gcf, 'PaperPosition', [0 0 0.25 0.15])
saveas(gcf, ['output_plots/', 'Bcross', '.svg']);
% Dilate
figure('visible', 'off');
imshow(IPdilate(f, Bcross));
saveas(gcf, ['output_plots/', imname, '_Bcross', '_dilated', '.svg']);
% Erode
figure('visible', 'off');
imshow(IPerode(f, Bcross));
saveas(gcf, ['output_plots/', imname, '_Bcross', '_eroded', '.svg']);

%% Square
Bsquare = true(3, 3);
figure('visible', 'off');
colormap(gray(256));
imagesc(Bsquare);
xticks(unique(round(get(gca, 'xTick')))); % only whole value tick labels
yticks(unique(round(get(gca, 'yTick')))); % only whole value tick labels
set(gcf, 'PaperUnits', 'normalized')
set(gcf, 'PaperPosition', [0 0 0.25 0.15])
saveas(gcf, ['output_plots/', 'Bsquare', '.svg']);
% Dilate
figure('visible', 'off');
imshow(IPdilate(f, Bsquare));
saveas(gcf, ['output_plots/', imname, '_Bsquare', '_dilated', '.svg']);
% Erode
figure('visible', 'off');
imshow(IPerode(f, Bsquare));
saveas(gcf, ['output_plots/', imname, '_Bsquare', '_eroded', '.svg']);

%% Big square
Bbigsquare = true(15, 15);
[r, c] = meshgrid(1:15, 1:15);
Bbigsquare(r + c < 7) = 0; % remove North-West corner
figure('visible', 'off');
colormap(gray(256));
imagesc(Bbigsquare);
xticks(unique(round(get(gca, 'xTick')))); % only whole value tick labels
yticks(unique(round(get(gca, 'yTick')))); % only whole value tick labels
set(gcf, 'PaperUnits', 'normalized')
set(gcf, 'PaperPosition', [0 0 0.25 0.15])
saveas(gcf, ['output_plots/', 'Bbigsquare', '.svg']);
% Dilate
```

```
figure('visible', 'off');
imshow(IPdilate(f, Bbigsquare));
saveas(gcf, ['output_plots/', imname, '_Bbigsquare', '_dilated', '.svg']);
% Erode
figure('visible', 'off');
imshow(IPerode(f, Bbigsquare));
saveas(gcf, ['output_plots/', imname, '_Bbigsquare', '_eroded', '.svg']);

% Compare with Matlab's built-in functions
%% Compare IPdilate to imdilate
figure('visible', 'off');
g = IPdilate(f, Bbigsquare);
imshow(g);
saveas(gcf, ['output_plots/', imname, '_Bbigsquare', '_IPdilate', '.svg']);
% imdilate
figure('visible', 'off');
g2 = imdilate(f, Bbigsquare);
imshow(g2);
saveas(gcf, ['output_plots/', imname, '_Bbigsquare', '_imdilate', '.svg']);
% difference
figure('visible', 'off');
imshow(g - g2);
saveas(gcf, ['output_plots/', imname, '_Bbigsquare', '_dilate', '_diff', '.svg
    ']);

%% Compare IPerode to imerode
figure('visible', 'off');
g = IPerode(f, Bbigsquare);
imshow(g);
saveas(gcf, ['output_plots/', imname, '_Bbigsquare', '_IPerode', '.svg']);
% imerode
figure('visible', 'off');
g2 = imerode(f, Bbigsquare);
imshow(g2);
saveas(gcf, ['output_plots/', imname, '_Bbigsquare', '_imerode', '.svg']);
% difference
figure('visible', 'off');
imshow(g - g2);
saveas(gcf, ['output_plots/', imname, '_Bbigsquare', '_erode', '_diff', '.svg'
    ]);

%% Erode with 3 different SE's - Figure 9.5 from the book.
% 11x11 square SE
figure('visible', 'off');
imshow(IPerode(f, true(11, 11)));
saveas(gcf, ['output_plots/', imname, '_11x11SE', '_erode', '.svg']);
% 15x15 square SE
figure('visible', 'off');
imshow(IPerode(f, true(15, 15)));
saveas(gcf, ['output_plots/', imname, '_15x15SE', '_erode', '.svg']);
% 45x45 square SE
figure('visible', 'off');
imshow(IPerode(f, true(45, 45)));
saveas(gcf, ['output_plots/', imname, '_45x45SE', '_erode', '.svg']);
```

A.2 Exercise 3

A.2.1 Exercise 3 (a)

Listing 5: IPskeletondecomp.m: Perform skeleton decomposition of the binary input image.

```
% IPskeletondecomp Decomposes a binary image into a skeleton.
%   Arguments:
%       A: input image. Must be binary, i.e. have logical values.
%       B: structuring element (SE). Must have odd dimensions and also be
%       binary, i.e. have logical values. Its origin is automatically set
%       at its centerpoint.
%   Returns: skeleton of the input image using the encoding from the
%   assignment
function sum_Sk = IPskeletondecomp(A, B)
    % (A erode k*B) - ((A erode k*B) dilate B)
    assert(islogical(A));
    assert(islogical(B));

    % Opening of input
    eroded_A = IPerode(A, B);
    opened_A = IPdilate(eroded_A, B);

    % Sk = input - opened input
    % (for all k>0 cases the input has been eroded in the previous
    % itteration of the recursion)
    Sk = A - opened_A;

    % Basecase: figure is empty after erosion
    if (sum(eroded_A, 'all') == 0)
        sum_Sk = Sk;
        return
    end

    % The next recursion will start with the eroded input of this recursion
    next_Sk = IPskeletondecomp(eroded_A, B);
    next_Sk(next_Sk~=0) = next_Sk(next_Sk~=0) + 1;
    sum_Sk = Sk + next_Sk;
end
```

A.2.2 Exercise 3 (b)

Listing 6: IPskeletonrecon.m: Perform image reconstruction using the skeleton.

```
% IPskeletonrecon Reconstructs a binary image from a skeleton.
%   Arguments:
%       skeleton: skeleton image. Must use the encoding from the
%       assignment and thus be of a type that allows for numbers > 1.
%       B: structuring element (SE). Must have odd dimensions and also be
%       binary, i.e. have logical values. Its origin is automatically set
%       at its centerpoint.
%   Returns: reconstructed image
function sum_Sk_recon = IPskeletonrecon(skeleton, B)
    assert(islogical(B));

    % Extract part from skeleton where values are 1
    Sk_recon = zeros(size(skeleton));
    Sk_recon(skeleton==1) = 1;
    Sk_recon = logical(Sk_recon);

    % Decrease all non-zero values by 1
    skeleton(skeleton~=0) = skeleton(skeleton~=0) - 1;

    % Base case: all skeletons have been processed
    if (sum(skeleton, 'all') == 0)
        sum_Sk_recon = Sk_recon;
        return
    end

    % Dilate the skeletons for all values of k
    k = max(skeleton, [], 'all');
    next_skel = zeros(size(skeleton));
    for i=1:k
        skel_k = zeros(size(skeleton));
        skel_k(skeleton==i) = 1;
        dilated_skel_k = IPdilate(logical(skel_k), B);
        if (i == 1)
            intermediate_skeleton = dilated_skel_k;
        end
        next_skel(dilated_skel_k==1) = i;
    end

    % Recursively call the next dilation step
    sum_Sk_recon = logical(Sk_recon + IPskeletonrecon(next_skel, B));
end
```

A.2.3 Exercise 3 (c)

Listing 7: IPskeleton_test.m: Test script to demonstrate the functionality of both IPSKELETONDECOMP and IPSKELETONRECON.

```
clc;                                    % clear the command window
close all;                              % close open figure windows
clear;                                  % clear workspace

%% Compute skeleton- and reconstruction of nutsbolts image
imname = 'nutsbolts';
inputfile = ['input_images/', imname, '.tif'];
f = imread(inputfile);                     % read input image

% Compute skeleton and the reconstruction
B_square = logical([1 1 1; 1 1 1; 1 1 1]);
skeleton = IPskeletondecomp(f, B_square);
reconstruction = IPskeletonrecon(skeleton, B_square);

% Calculate the difference between the input and the reconstruction
difference = sum(abs(double(f) - double(reconstruction)), 'all');

% Show input, skeleton, result and difference
figure;
subplot(221);
```

```
imshow(f)
title("input image");
subplot(222);
imshow(reconstruction)
title("reconstruction");
subplot(223);
imshow(skeleton)
title("IPskeletondecomp");
subplot(224);
imshow(f - reconstruction)
title("difference=" + difference);

% Save current plot
saveas(gcf, ['output_plots/', imname, '_skeleton_test.svg']);
exportgraphics(gcf, ['output_plots/', imname, '_skeleton_test.png']);

% Plot and save individual images
figure('visible', 'off');
imshow(f)
saveas(gcf, 'output_plots/skeleton_input.svg');
figure('visible', 'off');
imshow(reconstruction)
saveas(gcf, 'output_plots/skeleton_reconstruction.svg');
figure('visible', 'off');
imshow(skeleton)
saveas(gcf, 'output_plots/skeleton_skeleton.svg');
figure('visible', 'off');
imshow(f - reconstruction)
saveas(gcf, 'output_plots/skeleton_difference.svg');


%% Example from the slides (with borders added).
f = logical([   0 0 0 0 0 0 0;
                0 1 0 0 0 0 0;
                0 0 1 1 0 0 0;
                0 0 1 1 0 0 0;
                0 0 1 1 1 0 0;
                0 0 1 1 1 0 0;
                0 1 1 1 1 1 0;
                0 1 1 1 1 1 0;
                0 1 1 1 1 1 0;
                0 1 1 1 1 1 0;
                0 1 1 1 1 1 0;
                0 0 0 0 0 0 0;]);

skeleton = IPskeletondecomp(f, B_square);
reconstruction = IPskeletonrecon(skeleton, B_square);

% Show input, skeleton, result and difference
figure;
subplot(221);
imshow(f)
title("input image");
subplot(222);
```

```
imshow(reconstruction)
title("reconstruction");
subplot(223);
imshow(skeleton)
title("IPskeletondecomp");
subplot(224);
imshow(f - reconstruction)
title("difference=" + difference);
```