NTNU
Norwegian University of
Science and Technology

Digital Image Processing Assignment 3

Marius Blom

Sondre Jensen

2017

TDT4195 - Visual Computing Fundamentals

TABLE OF CONTENTS

1 THEORY QUESTIONS

a Segmentation in computer vision

The main goal with segmentation is to separate the foreground object from the background. We do this to focus on the object instead of the background. This can be difficult since we try to design algorithms which mimic the human visual system, which have evolved over thousands of years. Our brain have models of the world and is therefore able to make adjustments as to what to focus on, something that is hard to replicate with a computer. When it comes to segmentation in image processing, we have no control of the environment. This can lead to objects that are close to each other can overlap or objects getting mixed into the background. Segmentation is also very vulnerable to noise, which makes it much harder to separate different objects.

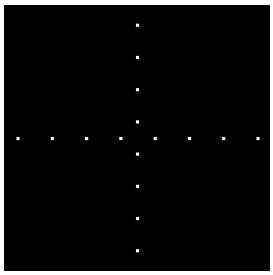
b Applying the Hough transform

We can see from image 1.1b that we have four strong lines in four different directions. This show with the four white dots where the lines meet in the Hough image. Since the lines form a common pattern, this indicates that we have an intact object on the original image. Thus we can conclude that image A and image 3 belong together.

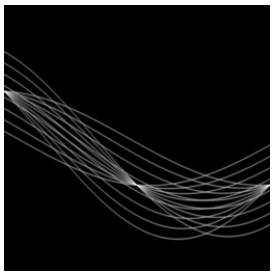
(a) Image A **(b)** Image 3

Figure 1.1: A3

We can see from image 1.2b that we can connect with image 1.2a. We have a total of 16 lines in image 7 that represent the 16 dots in image B. We can see that we are able to draw two lines through the original image since all the lines intersect in the Hough image. The horizontal line repeats at the start and at the end of image 7. Thus we can conclude that image B and image 3 belong together.



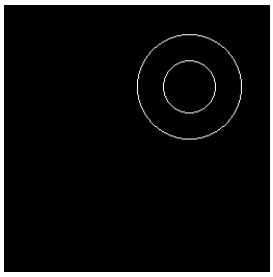
(a) Image B



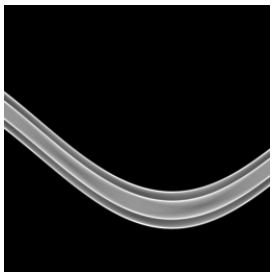
(b) Image 7

Figure 1.2: B7

We can see from image 1.3a and image 1.3b that it is easy to identify that image C and image 5 belong together, since we have two identical areas in image 5. It is clear that every line we draw through the smallest object will also be drawn through the largest object since the one area in image 5 is completely surrounded by the other, hence one object is inside the other. Thus we can conclude that image C and image 5 belong together.



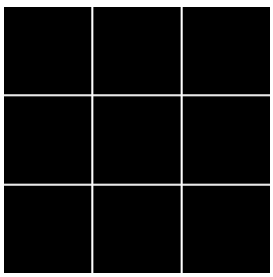
(a) Image C



(b) Image 5

Figure 1.3: C5

We can see from image 1.4a that we have two vertical lines and two horizontal lines in the original image, much in the same manner as with figure 1.2. We can also see that we have intersections at the edge of the image, this represents the horizontal lines which repeat at the end of image 6. Thus we can conclude that image D and image 6 belong together.

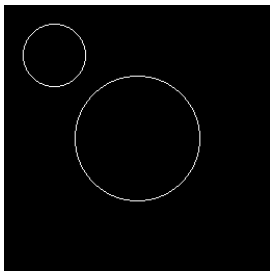


(a) Image D

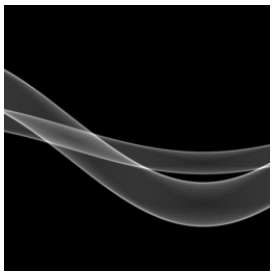
(b) Image 6

Figure 1.4: D6

We can see from image 1.5b that we have two objects due to the outlines. We can also see that only in a small part of the image are we able to draw a line through both the objects. Since the area that indicates the objects are smooth can we identify them as circles, since we are not able to form lines out of circles. This meas that image E and image 1 belong together.



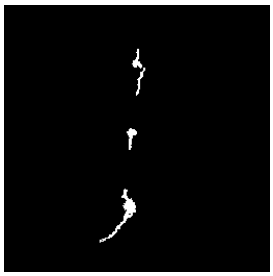
(a) Image E



(b) Image 1

Figure 1.5: E1

We can see from image 1.6b that we have three small objects that are able to form a line at about the centre of the Hough image, representing a vertical line. This description resembles image F and therefor must image F and image 2 belong together.

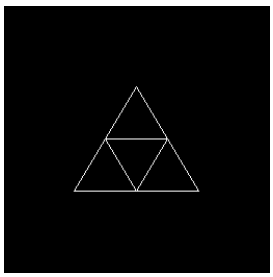


(a) Image F

(b) Image 2

Figure 1.6: F2

We are now left with image image G and image 4 by the process of elimination. We can see that we are able to form 6 lines out of image 4, and two of them are horizontal as they repeat at the start and at the end. Since two and two intersections are on top of each other in image 4 can we see that the pair of lines have the same angle in the original image. This results in the figure depicted in image G.



(a) Image G

(b) Image 4

Figure 1.7: G4

c Morphological operations, linear?

Morphological image processing is a collection of non-linear operations related to the shape or morphology of features in an image. Binary morphological operators may be implemented using convolution-like algorithms with the fundamental operations of addition and multiplication replaced by logical OR and AND.

d Determine erosion

The erosion of the 6x5 binary picture is shown in figure 1.8.

0	0	0	0	0
0	1	0	0	0
0	1	1	0	0
0	0	1	1	0
0	0	0	1	0
0	0	0	0	0

Figure 1.8: Eroded 6x5 picture

2 Region Growing

a Segment a grayscale

We loaded the original grayscale image in MATLAB to detect which seed-points to use. On figure 2.1 can we see the points elected as seed-points.

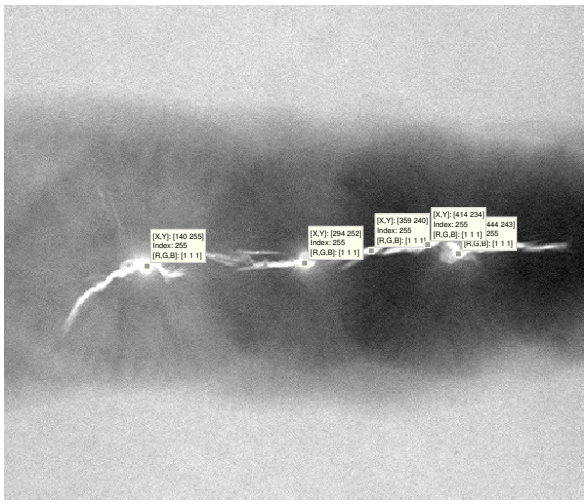


Figure 2.1: Selected points

We elected to use Moore neighbourhood to map the selected area as this in our minds would lead to best mapping.

```
def flood_fill(image, possitionX, possitionY, seedPointX, seedPointY, thr
if (possitionX < 1 or possitionY < 1):
return canvas
if (image[possitionY,possitionX] == canvas[possitionY,possitionX]):
return canvas
if (np.abs(image[possitionY, possitionX]-image[seedPointY, seedPointX])<t
canvas[possitionY, possitionX] = image[possitionY, possitionX]
canvas = flood_fill(image, possitionX-1, possitionY-1, seedPointX, seedPo
canvas = flood_fill(image, possitionX, possitionY-1, seedPointX, seedPoint
canvas = flood_fill(image, possitionX+1, possitionY-1, seedPointX, seedPo
canvas = flood_fill(image, possitionX-1, possitionY, seedPointX, seedPoint
canvas = flood_fill(image, possitionX, possitionY, seedPointX, seedPointY
```

```
canvas = flood_fill(image, possitionX+1, possitionY, seedPointX, seedPoint
canvas = flood_fill(image, possitionX-1, possitionY+1, seedPointX, seedPo
canvas = flood_fill(image, possitionX, possitionY+1, seedPointX, seedPoint
canvas = flood_fill(image, possitionX+1, possitionY+1, seedPointX, seedPo
return canvas
```

i Region growing implementation

We used the following points: (140, 255) (294, 252) (359, 240) (414, 234) (444, 243)

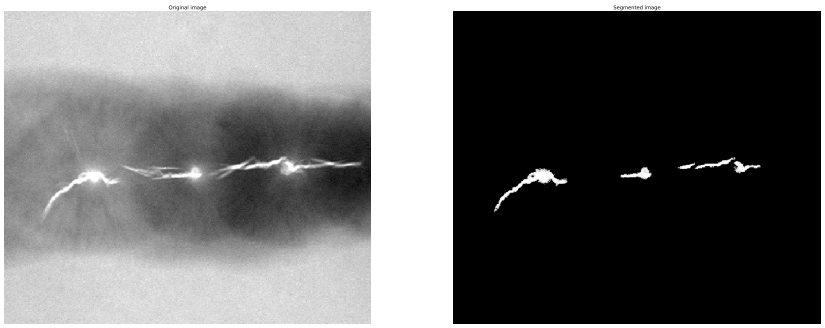


Figure 2.2: Point stuff

As can be seen from figure 2.2, have the area connected to the point been highlighted.

3 Noise Removal in Binary Images

a Erosion, dilation, opening and closing

In this task have we coded the erosion and dilation operations manually. This made the run time really slow, but felt it was necessary in order to show that we know how these two operations work.

Since the opening operation consists of erosion followed by dilation and closing operation consists of dilation followed by erosion, have we used these terms to explain how we removed the noise.

```
def makeBinaryImage(image, threshold):
        yImage = np.size(image, 0)
        xImage = np.size(image, 1)
        binaryImage = np.zeros((yImage,xImage), dtype = 'bool')
        for y in range(yImage):
                for x in range(xImage):
                        if(image[y,x] > threshold):
                                binaryImage[y,x] = 1
        return binaryImage
```

```
def createKernel(size):
kernel = np.full((size,size),0)
kernel = kernel.astype(bool)
centerKernel = size//2
for y in range(size):
for x in range(size):
r = np.sqrt(np.power(x-centerKernel,2) + np.power(y-centerKernel,2))
if (r<centerKernel):
        kernel[y,x] = 1
return kernel
```

```
def erosion(image, kernel):
yImage = np.size(image, 0)
xImage = np.size(image, 1)
print('start Erosion')
erodedImage = np.full((yImage,xImage),0)
erodedImage = erodedImage.astype(bool)
for y in range(yImage):
for x in range(xImage):
        erodedImage[y,x] = travaserErosion(image, x, y, kernel)
```

```
return erodedImage
```

```
def travaserErosion(image, x, y, kernel):
yImage = np.size(image, 0)
xImage = np.size(image, 1)
sizeKernel = np.size(kernel, 0)
centerKernel = sizeKernel//2
for j in range(sizeKernel):
for i in range(sizeKernel):
if(kernel[j,i]):
        currentPixelY = y - j + centerKernel
        currentPixelX = x - i + centerKernel
        if (0 <= currentPixelY < yImage and 0 <= currentPixelX < xImage )
                if(image[currentPixelY, currentPixelX] == 0):
                        return 0
return 1
```

```
def dilation(image, kernel):
yImage = np.size(image, 0)
xImage = np.size(image, 1)
print('start dilation')
dilatedImage = np.full((yImage,xImage),0)
dilatedImage = dilatedImage.astype(bool)
for y in range(yImage):
        for x in range(xImage):
                dilatedImage[y,x]=traverseDilation(image, x, y, kernel)
return dilatedImage
```

```
def traverseDilation(image, x, y, kernel):
yImage = np.size(image, 0)
xImage = np.size(image, 1)
sizeKernel = np.size(kernel, 0)
centerKernel = sizeKernel//2
for j in range(sizeKernel):
for i in range(sizeKernel):
if(kernel[j,i]):
        currentPixelY = y - j + centerKernel
        currentPixelX = x - i + centerKernel
        if (0 <= currentPixelY < yImage and 0 <= currentPixelX < xImage )
                if(image[currentPixelY, currentPixelX] == 1):
                        return 1
return 0
```

i Noise removal process

We started with an opening operation in order to remove all the noise from outside the figures. Afterwords did we use a closing operation in order to fill the empty holes inside the figures. We elected to use a circular kernel in this task, as it was the least destructive towards the figures. The kernel we used can be seen in figure 3.1

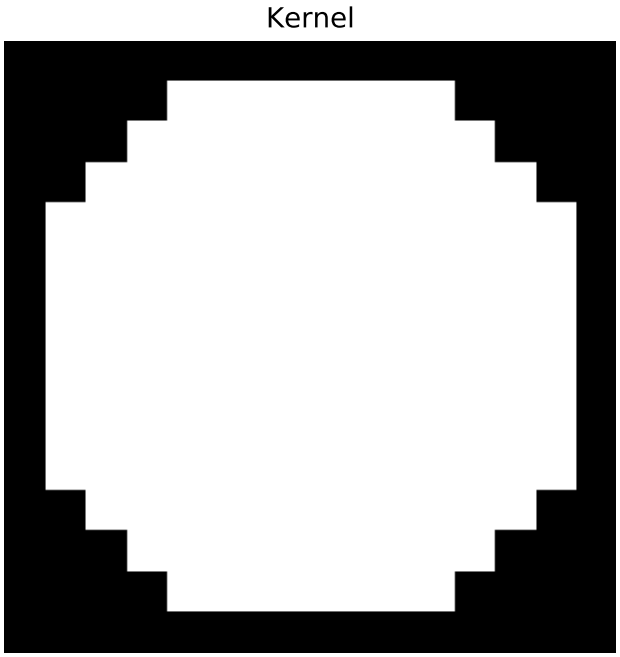


Figure 3.1: Task03, Kernel

ii Result

The results from this task can be shown on Figure 3.2. We can easily see how the opening operation removes the noise outside the figures, and the closing operation removes the noise inside the figures.

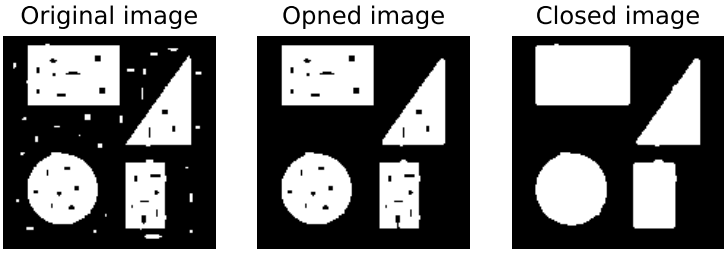


Figure 3.2: Plots of the result

4 Boundary Extraction

a Extract the boundary

By implementing code a that eroded a 3x3 kernel of ones, at the noise-free image from task 3 where we able to extract a slightly smaller image. By then removing this eroded image from the original noise-free image were we left with the borders of the figures.

```
def boundary(image, erodedImage):
        yImage = np.size(image, 0)
        xImage = np.size(image, 1)
        boundaryImage = np.full((yImage,xImage),0)
        boundaryImage = boundaryImage.astype(bool)
        for y in range(yImage):
                for x in range(xImage):
                        if(image[y,x]==erodedImage[y,x]):
                                boundaryImage[y,x]=False
                        else:
                                boundaryImage[y,x]=True
        return boundaryImage
```

On figure 4.1 can we see the original- and the border-image.

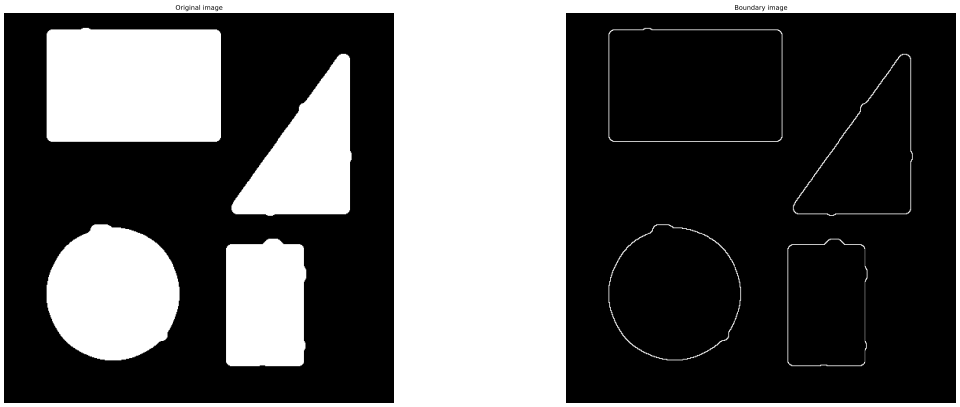


Figure 4.1: Point stuff

5 Shape Recognition

The code we implemented uses samples from the different square in the chessboard to calculate the average colour of the green and the purple areas. We thereafter navigate through the entire image examining the colour values of each pixel to see if it is close to this two values. If the colours deviates from purple or green will the area be marked white, otherwise will it be marked black.

After doing this we were left with some lines resembling the border of the green and purple areas. These were removed using a opening followed by a closing operation, with a 3x3 kernel consisting of ones.

a Remove chessboard

The result from this operation can be seen on figure 5.3

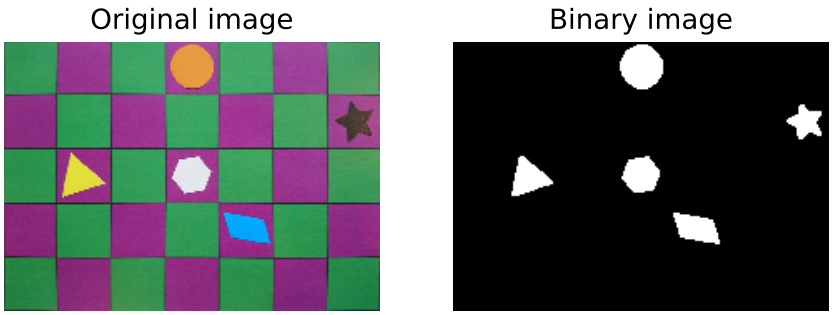


Figure 5.1: task5-01

As we can see on figure 5.2 and figure 5.3, this algorithm works for all the images given in this assignment.

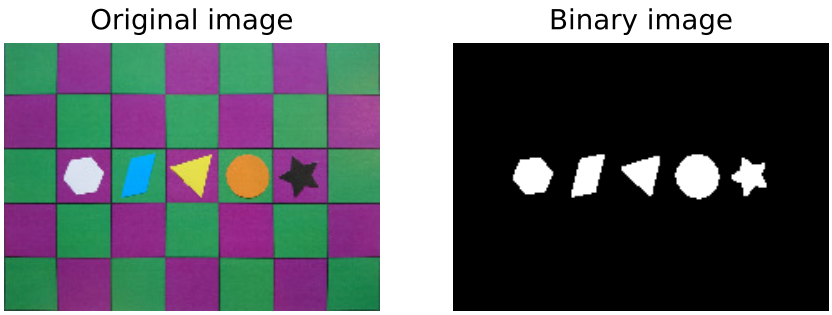


Figure 5.2: task5-02

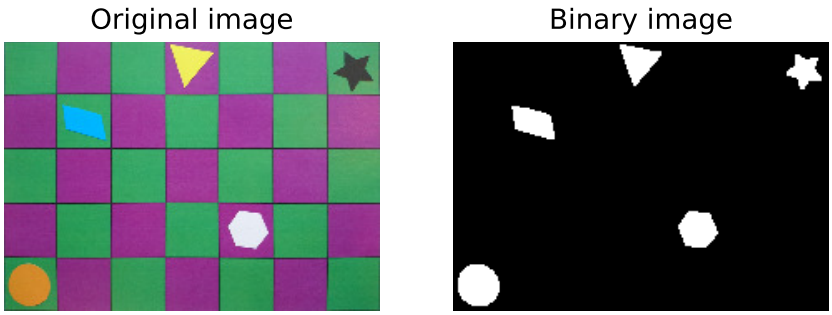


Figure 5.3: task5-03

b Label objects

This code is the code we used to achieve labeling. It solves the problem of recursive labeling in a bad way, by traversing the image again, hence the bad run time. If you plan on running it to check if it works, please do start the code before you go to lunch or something.

```
#       this is not optimal, this function is REALLY slow
def findFigure(image):
yImage = np.size(image, 0)
xImage = np.size(image, 1)
canvas = np.full((yImage,xImage),0)
canvas.astype('int')
label=0
for y in range(yImage):
        for x in range(xImage):
        if (image[y,x]==0):
                canvas[y,x]=0
        else:
                if(canvas[y-1,x]==0 and canvas[y,x-1]==0):
                        canvas[y,x]=label+ 1
                elif(canvas[y-1,x]!=0 and canvas[y,x-1]==0):
                        canvas[y,x]=canvas[y-1,x]
                elif(canvas[y,x-1]!=0 and canvas[y-1,x]==0):
                        canvas[y,x] = canvas[y,x-1]
                elif(canvas[y,x-1]!=0 and canvas[y-1,x]!=0):
                        canvas[y,x]=canvas[y-1,x]
                        label=canvas[y,x]
                        temp=canvas[y,x-1]
                        for tempY in range(yImage):
                                for tempX in range(xImage):
                                        if(canvas[tempY,tempX]==temp):
                                                canvas[tempY,tempX]=label
                        label=canvas[y,x]
return canvas
```

We were able to achieve labeling by coding the label algorithm by hand. This turned out to be REALLY SLOW, since it iterates the image more than is necessary. We did not have time to fix it to achieve a better runtime. We can easily achieve the same result by using the line below. The code we delivered is using ndimage.label and our own function is commented out.

```
labeled, nr_objects = ndimage.label(binaryImage)
```

In the end the result image, as seen on figure 5.4, have achieved the objective and completed the task. Even though the run time is horrible.

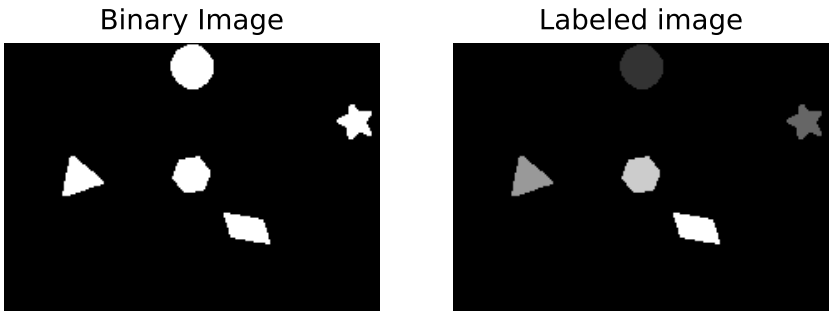


Figure 5.4: task5-03

c List of all the shape types

D.N.F.