ME5405: Machine Vision

Group Report

Digital Image Processing using MATLAB

Student Names:	*Student Number:*
Arijit Dasgupta	**A0182766R**
Chong Yu Quan	**A0136286Y**

Student Emails:

arijit.dasgupta@u.nus.edu

e0321496@u.nus.edu

November 9, 2020

Contents

List of Figures

Acknowledgements

We would like to express our gratitude to Dr Chui Chee Kong and Dr Lim Kah Bin for teaching us the fundamentals of Machine Vision in this module, ME5405. We have thoroughly enjoyed learning from the lectures and have found a great deal of help from the notes provided to us. We were able to conduct most of the project by relying on what we have learnt from the lecture materials of this module. We also thoroughly enjoyed doing this project from scratch, and we are proud that we did not have to rely on GitHub or any senior for help. This project was our first time programming in MATLAB, and we have gained a lot from the experience.

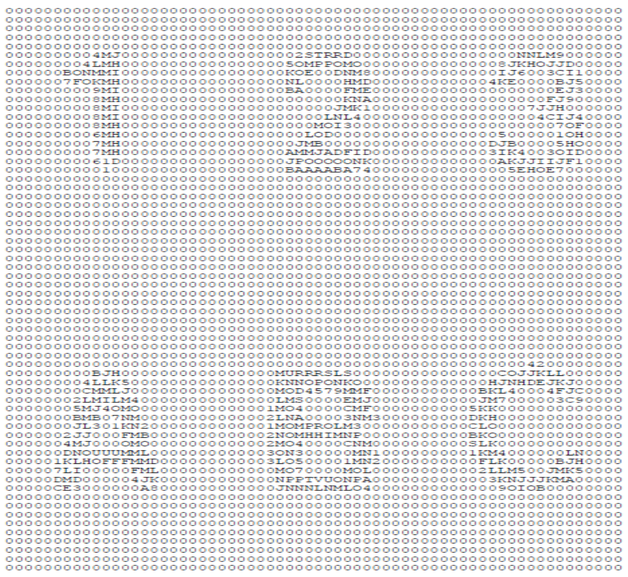
We will certainly recommend this module to our peers and juniors, so that they can experience going through the meaningful work for this project, just like we did.

Chapter 1

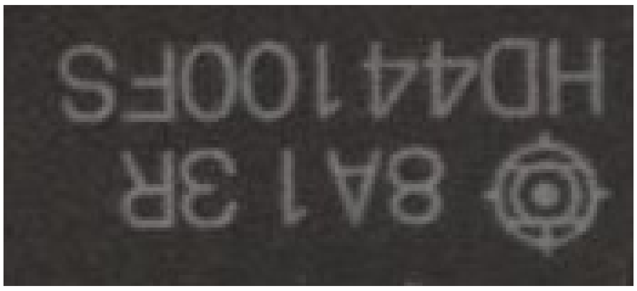
Introduction

This report elaborates on the image processing tasks to process the images shown in Figure 1.1. Image 1 shown in Figure 1.1a is a 64×64, 32 quantized-level image in the format of a text document. It consists a coded array that contains an alphanumeric character for each pixel in the image. The alphanumeric characters ranges from 0-9 and A-V, corresponding to the 32 gray levels where 0 represents black and V represents white in grayscale. Image 2 shown in Figure 1.1b is a JPEG image of a label on a microchip with its characters inverted. The characters are required to be inverted before proceeding with any further image processing. The following list states the imaging processing steps for the two images that are to be performed sequentially.

1. Display the original image on screen
2. Create a binary image using thresholding.
3. Segment the image to separate and identify the different characters
4. Rotate the characters in the image about their respective centroids by 90° clockwise
5. Rotate the characters in the image from Step 4 about their respective centroids by 35° counter-clockwise
6. Determine the outline(s) of characters of the image
7. Determine a one-pixel thin image of the characters
8. Arrange the characters with the sequence: **A1B2C3** for Image 1 and **81344100ARHDFS** for Image 2



(A) Image 1



(B) Image 2

Figure 1.1: Original Images for processing

The entire image processing was done in MATLAB 2019 [1] with the entire code being written from scratch. **A great deal of effort was made to code most of the image processing steps manually, without relying on any built-in MATLAB functions**. This was done so that we could become more accustomed and familiar with all aspects of the image processing algorithms. These low-level implementations and algorithms were done with a lot of pixel-level point processing and the use of MATLAB array and cell array data structures. The following explicitly list all the algorithms and implementations that were manually coded from scratch despite the availability of simple MATLAB built-in functions that fulfil the same task:

- Inverting Image 2 by switching the indices for each pixel in the image.
- Manually generating the image histogram values by scanning through all pixels
- Point-processing binary thresholding operation using an IF-ELSE condition for all pixels
- Manually coding out the entire iterative algorithm as mentioned in the ME5405 Part 2 notes with an iterative set of top-down, left-right and bottom-up, right-left scans to label connected objects until the labels show no change
- Manually coding out the classical algorithm from the ME5405 Part 2 notes that did one top-down, left-right scan of the image to assign labels and save all equivalences. The equivalences were used as instructions for a union-find for disjointed set graph-search algorithm that we also

manually implemented in MATLAB using a LabelNode handle class that we coded as well. This allowed us to conduct one more scan through the image to relabel the characters.

- Manually implementing the algorithm to use the labels from the segmentation algorithms to separate all the characters into separate images saved in a cell array.
- Manually programmed the algorithm to locate the centroid of the images as well as the algorithm for linear, bilinar & bicubic brightness interpolation for the rotation of an image for any specified angle. This was possible by coding the backward mapping of coordinates via the rotation matrix for all discrete points in the image.
- Manually programmed the algorithms for Butterworth & Gaussian filtering in the spatial frequency domains, aided by the MATLAB's built-in Fast-Fourier transform function. This was done to determine the outline of the image.
- Using MATLAB's built-in structured element objects and erode function for mathematical morphology, we manually coded out a hit-and-miss iterative thinning algorithm to create pixel-thin representations of the characters.

In the next chapter, we will explore the different image post-processing steps for both images, along with the different approaches for each step.

Chapter 2

Image Processing Workflow

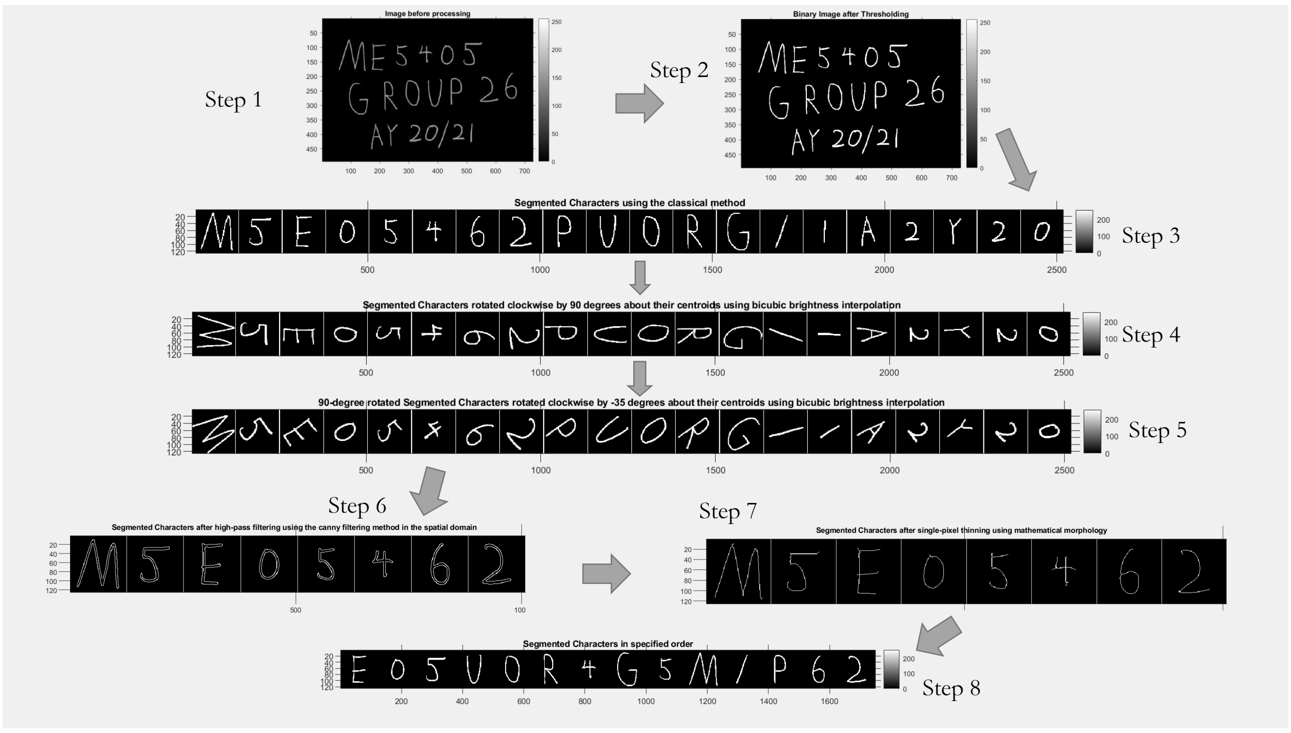

Step 1
Image before processing
Step 2
Binary Image after Thresholding
Step 3
Segmented Characters using the classical method
Step 4
Segmented Characters rotated clockwise by 90 degrees about their centroids using bicubic brightness interpolation
Step 5
90-degree rotated Segmented Characters rotated clockwise by -35 degrees about their centroids using bicubic brightness interpolation
Step 6
Segmented Characters after high-pass filtering using the canny filtering method in the spatial domain
Step 7
Segmented Characters after single-pixel thinning using mathematical morphology
Step 8
Segmented Characters in specified order

FIGURE 2.1: Images Processing Flowchart, using a custom image as an example

Figure 2.1 highlights the image processing workflow for this project. The flowchart is based on a custom handwritten image that we inserted into our code. The first step simply shows the grayscale image before processing. The next step converts the image into a binary thresholded image. The third step segments the image into its respective characters. The fourth and fifth steps involves the rotation of each character about its centroid, 90 ° clockwise and 35 ° counterclockwise respectively. The sixth step uses a form of edge detection to create an image with the outline only. Step seven then creates a pixel-thin representation of each character. In Figure 2.1, only a portion of the characters are shown for steps six and seven, to clearly show the outlines and pixel-thin characters. In the final step, the characters are arranged in a user-specified order.

2.1 Display Original Image

The first step of the image processing requires us to display the image. However this is not as simple as simply showing the image. In the case of Image 1, the image is provided mainly in the range of 0-9 & A-V, which basically quantizes the gray-level brightness into 32 levels. To view this image in grayscale in MATLAB, the 32 levels are converted into integer value representation, and stretched to a 0-255 scale, like that of Image 2. The conversion is represented in Equation 2.1 below.

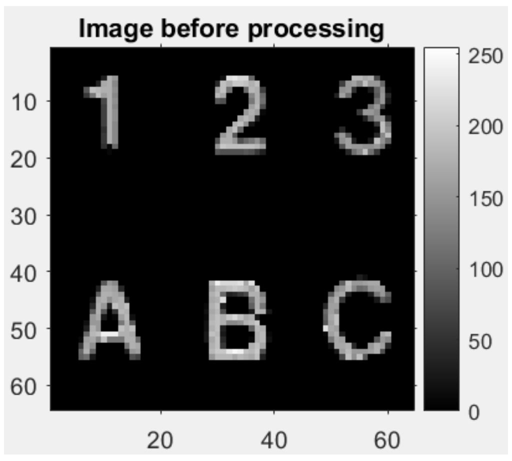
$$S = 8(R - 1) \tag{2.1}$$

R in the equation refers to the value of the original quantized value mapped to an integer from 1 to 32. Hence 0-9 is mapped to 1-10 while A-V is mapped to 11-32. S refers to the final brightness value for the grayscale Image 1. This means that background values that were originally 0, remain as 0 in the new scale. Likewise, the brightest value of V is converted to 248 in the image. Although one, would expect this value to represent the brightest possible (255), it will be resolved after binary thresholding. Image 1 is then displayed as in Figure 2.2a.

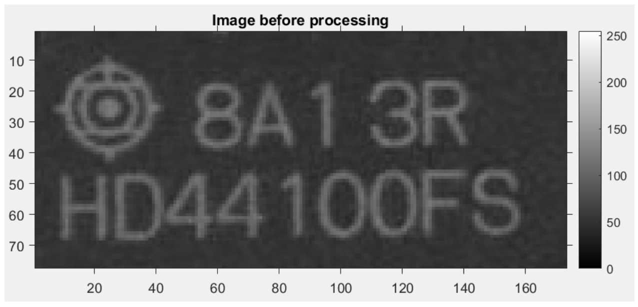
In the case of Image 2, as per instructions, the entire image is inverted by swapping the indices at each pixel with the respective position for inversion. The algorithm for this is described in pseudo-algorithm 1 as shown below, which directly manipulates the brightness values on the discrete raster and does not use any form of interpolation. Image 2 is then displayed as in Figure 2.2b.

Pseudo Algorithm 1: Image Inversion

```
Existing Variable(s):
    • Image array $I(x, y)$
Initialise:
    • Temp Image array $I^*(x, y) \leftarrow$ zeros
    • $R \leftarrow$ Number of Rows in $I(x, y)$
    • $C \leftarrow$ Number of Columns in $I(x, y)$
Execute:
    • For each pixel, in row $x$ and column $y$, in $I(x, y)$:
        – $I^*(R - x + 1, C - y + 1) \leftarrow I(x, y)$
    • $I(x, y) \leftarrow I^*(x, y)$
```



(A) Image 1 (after conversion to grayscale in 0 - 255)



(B) Image 2 (after inversion)

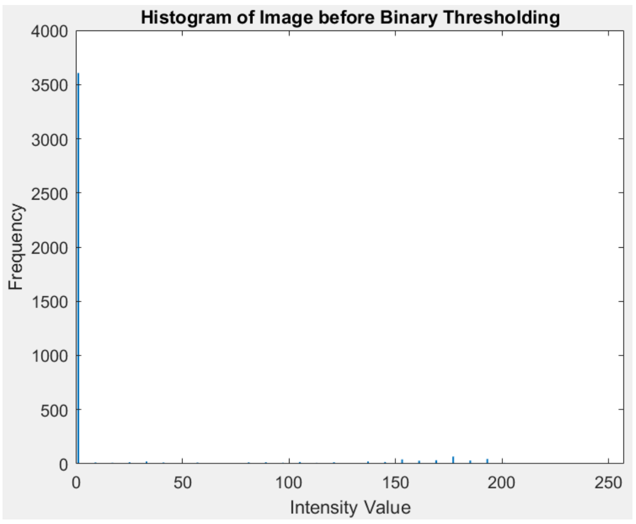
FIGURE 2.2: Images ready for processing

2.2 Binary Thresholding

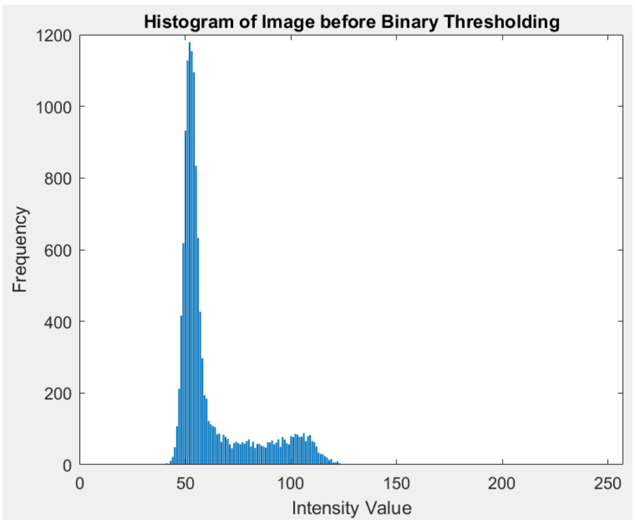
Binary thresholding is a type of point processing method where transformation is defined pixel by pixel. The transformation process occurs with no consideration of pixel location, being solely dependant on the grey-level intensity value for this application. It is implemented using a simple nested for loop structure to iterate over each row and and column of pixels in the image to check if the intensity value exceeds a specified grey-level intensity value threshold. Using a simple IF-ELSE condition, pixels with grey-level intensity value exceeding or equal to the threshold will be assigned the largest grey-level intensity value (255), i.e. maximum illumination, while pixels with grey-level intensity value lesser than the threshold will be be assigned the smallest grey-level intensity value (0), i.e. no illumination. Pseudo-algorithm 2 shown below succinctly illustrates binary thresholding process.

Pseudo Algorithm 2: Binary Thresholding

```
Existing Variable(s):
    • Image array $I(x, y)$
Initialise:
    • Grey-level intensity threshold $T$
Execute:
    • For each pixel, in row $x$ and column $y$, in $I(x, y)$:
        – If $I(x, y) > T$: $I(x, y) \leftarrow 255$,
        – Else If $I(x, y) \leq T$: $I(x, y) \leftarrow 0$
```



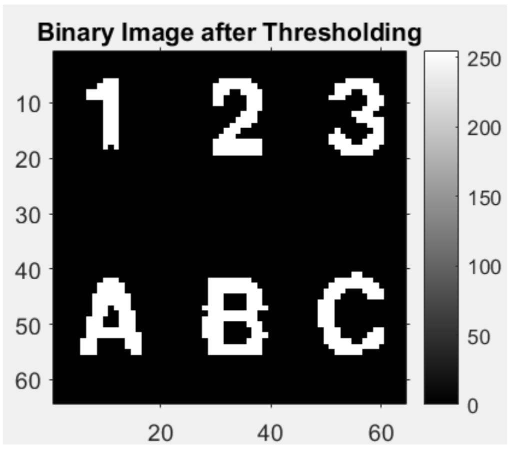
(A) Image 1



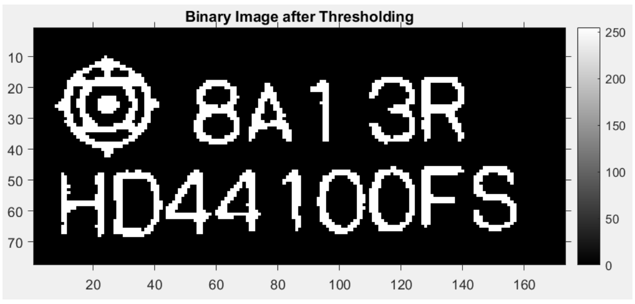
(B) Image 2

FIGURE 2.3: Histogram of Image

Figure 2.3a and 2.3b shows the histogram for Image 1 and 2 respectively. By observing the distribution of the histograms for each respective images, the grey-level intensity threshold can be selected by choosing a value in between the two peaks in the histogram for a bi-modal histogram. Following this principle, the grey-level intensity threshold for Image 1 and 2 was chosen to be 10 and 85 respectively. The results from the binary thresholding are illustrated in Figure 2.4 shown below.



(A) Image 1



(B) Image 2

FIGURE 2.4: Image after Binary Thresholding

2.3 Image Segmentation

Image segmentation, in essence, is a form of connected component analysis. In the case of a binary image, we carry out a connected component labeling of the binary 1 (or 255 after binary thresholding) pixels according to the type of connectivity. A label can be defined as a unique name or index of the region to which the pixels belong, i.e. the identifier for a potential object region. Connected component labelling is hence a grouping operation that makes a unit change from pixel to a region, of which comprises various types of useful properties (e.g. shape, centroid location and statistical grey-level intensity values).

Formally, two binary 1 pixels, $\mathbf{x}$ and $\mathbf{y}$, belong to same component $\mathbf{C}$ if there is a sequence of binary 1 pixels $\{\mathbf{x_0}, \mathbf{x_1}, \ldots \mathbf{x_n}\}$ where $\mathbf{x_0} = \mathbf{x}$, $\mathbf{x_n} = \mathbf{y}$ and $\mathbf{x_i}$ is a neighbour of $\mathbf{x_{i-1}}$. The type of connectivity is most succinctly defined in Figure 2.5 shown below of 4-connectivity and 8-connectivity, where the corresponding adjacent pixels illustrated for the two types of connectivity are considered neighbours.

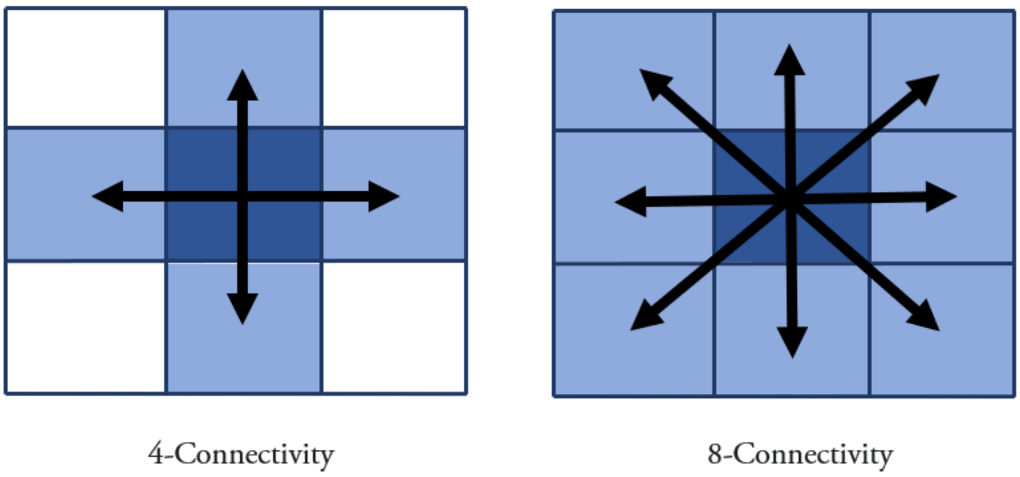


FIGURE 2.5: Pixel Connectivity

In this report, two connected component algorithms are utilised, namely the iterative & classical [2] algorithms. The stated algorithms both adopt the 4-connectivity notion of connectivity for image segmentation and outputs a symbolic image in which the label assigned to each pixel is an integer uniquely identifying the connected component to which that pixel belongs in the original input binary image. The first algorithm, the iterative algorithm, essentially consists of 3 key steps. In the initialisation step, each pixel with grey-level intensity of 255 is given a unique non-zero integer label, after which a top-down pass, from the first row to the last row of the image, is executed. On each row, a left to right pass occurs, for which the value of each non zero pixel is replaced by the minimum value of its non-zero neighbours in a recursive manner. Given the top-down left-right propagation, the neighbouring pixels in this propagation are defined as the pixel above and on the left of the current pixel. Upon reaching the last pixel at the bottom-right of the image, a similar bottom-up right-left pass is executed till the top-left pixel is reached, with the difference being that the neighbouring pixels in this propagation are defined as the pixel below and on the right of the current pixel. These two propagation steps are defined as one iteration in the iterative algorithm and is repeated till there are no changes in labels are detected in the image to be generated as output. Pseudo-algorithm 3 shown below succinctly illustrates the iterative algorithm.

Pseudo Algorithm 3: Image Segmentation (Iterative Algorithm)

```
Existing Variable(s):
  • Image array $I(x, y)$
Initialise:
  • Temp Image array $I^*(x, y) \leftarrow$ Unique non-zero integer labels for pixels with grey-level intensity of 255 in $I(x, y)$ and zero otherwise
Execute:

While there are still changes to $I^*(x, y)$ in previous iteration
  • For each pixel, in row $x$ and column $y$ in increasing order, in $I^*(x, y)$:
    – If $I(x, y) = 255$:
      ∗ If $I(x-1, y) = 255$ or $I(x, y-1) = 255$ or both:
        · $I^*(x, y) \leftarrow \min(I^*(x-1, y), I^*(x, y-1))$, given that $I^*(x-1, y) \neq 0$ and $I^*(x, y-1) \neq 0$
  • For each pixel, in row $x$ and column $y$ in decreasing order, in $I^*(x, y)$:
    – If $I(x, y) = 255$:
      ∗ If $I(x+1, y) = 255$ or $I(x, y+1) = 255$ or both:
        · $I^*(x, y) \leftarrow \min(I^*(x+1, y), I^*(x, y+1))$, given $I^*(x-1, y) \neq 0$ and $I^*(x, y-1) \neq 0$
Return $I^*(x, y)$
```

Using the iterative algorithm, Image 1 and Image 2 can be processed with 4 and 5 iterations respectively. It can be observed that the iterative algorithm does not keep track of equivalences of the connected regions, instead relying on the number of iterations to complete the labelling. Hence, the iterative algorithm is useful in circumstances where hardware memory is severely constraint as it does not require any additional memory storage to track equivalences. On the other hand, the classical algorithm only requires a single iteration (2 passes), at the expense of storing a large equivalence table. The classical algorithm comprises of 3 key steps. First, a top-down left-right label propagation is executed like that in the previous algorithm without the need for an initialisation of non-zero integer values. IF the pixel is seen to have no non-zero neighbours, it will be given a unique label. IF it has non-zero neighbours, it will take the label of the minimum value. However, in addition to replacing the pixel value by the minimum value of its non-zero neighbours (based on 4-connectivity), the equivalence of the two non-zero labels from the neighbours are logged. Upon reaching the last pixel at the bottom-right of the image, the all the necessary equivalences amongst the labels would have been generated after the first pass. The second step involves using a union find of disjointed set graph search algorithm to resolve the various equivalences logged to derive the necessary equivalent classes. Finally, a second pass of top-down left-right label propagation is executed to relabel all the labels from the first pass with labels derived from the equivalent classes. Pseudo-algorithm 4 shown below succinctly illustrates the classical algorithm.

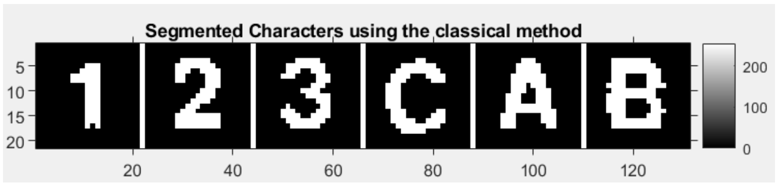
Pseudo Algorithm 4: Image Segmentation (Classical Algorithm)

```
Existing Variable(s):
  • Image array $I(x, y)$
Initialise:
  • Temp Image array $I^*(x, y) \leftarrow$ zeros
  • label counter $\leftarrow 0$
  • Equivalences list $\mathbf{E}$ storing tuple of equivalences $E_i$
  • Node class with following attributes:
    – Parent Node: Stores parent node (self-reference is possible) for which the current label is equivalent to.
    – Size: Stores number of child labels associated with current label (inclusive of itself)
    – Label: Current label
  • Nodes list $\mathbf{N}$ storing unique nodes, $N_i$, representing every unique label. Each node initialised using Node class, with Parent Node $\leftarrow$ itself, Size $\leftarrow 1$ and Label $\leftarrow$ current node's unique label
Execute:
  • For each pixel, in row $x$ and column $y$, in $I^*(x, y)$:
    – If $I(x, y) = 255$:
      ∗ If $I(x-1, y) = 255$ or $I(x, y-1) = 255$ or both:
        · $I^*(x, y) \leftarrow \min(I^*(x-1, y), I^*(x, y-1))$, given that $I^*(x-1, y) \neq 0$ and $I^*(x, y-1) \neq 0$
        · $E \leftarrow$ Append label equivalence of $[I^*(x-1, y), I^*(x, y-1)]$, given that $I^*(x-1, y) \neq 0$ and $I^*(x, y-1) \neq 0$
      ∗ Else: increment label counter & $I^*(x, y) \leftarrow$ label counter
```

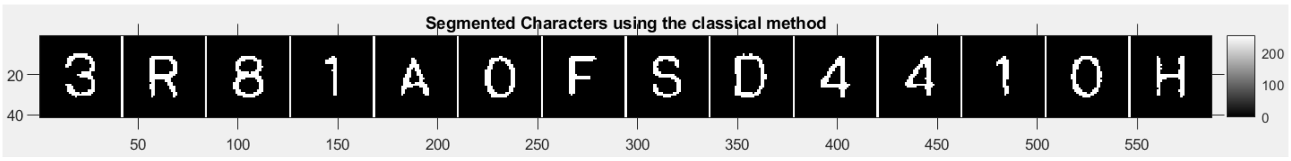
Pseudo Algorithm 4 (cont): Image Segmentation (Classical Algorithm)

```
• For each equivalence $E_i$ in $\mathbf{E}$:
    – Retrieve nodes, $N_x$ and $N_y$, from $\mathbf{N}$ with the corresponding label attribute as the label tuple in $E_i$
    – Recursively find root node of $N_x$ and $N_y$ from $\mathbf{N}$, $R_x$ and $R_y$, where parent node attribute of $R_x$ and $R_y$ is itself.
    – If $R_x \neq R_y$:
        ∗ The root node with smaller size attribute has its parent node attribute assigned the root node with larger size attribute.
        ∗ If the size attribute of both root nodes are equal, a root node is arbitrarily chosen to be to have its parent node attribute be assigned the other root node.
        ∗ The unassigned root node will increase its size attribute by the size attribute of the assigned root node.
• For each pixel, in row $x$ and column $y$ in $I^*(x, y)$:
    – If $I^*(x, y) \neq 0$:
        ∗ Retrieve node $N_i$ from $\mathbf{N}$ with the corresponding label attribute $I^*(x, y)$
        ∗ Recursively find root node of $N_i$, $R_i$, from $\mathbf{N}$.
        ∗ $I^*(x, y) \leftarrow$ Label attribute of $R_i$
Return $I^*(x, y)$
```

From the pseudo-algorithm, it is evident that the drawback of the classical algorithm is the memory space required for the storage of the equivalences, which may become untenable for larger and more complex images. However, for the relatively smaller and simpler Image 1 and 2, the classical algorithm can be readily executed without consuming significant memory resources in modern laptops. Figure 2.6 below showcases the various segmented characters displayed individually for both Image 1 and Image 2. It is observed than the results for both the iterative and classical algorithm are both identical after numerous runs, highlighting that the consistency and convergence of the two algorithms.



(A) Image 1



(B) Image 2

FIGURE 2.6: Segmented Images using the Iterative/Classical Algorithm

2.4 Image Rotation about Centroid

In this segment of the project, the aim is to take each one of the segmented images, identify its centroid and rotate the image about that point by a specified angle. The rotation involves the use of the rotation matrix on a backward mapped image. The brightness is then determined by an interpolation algorithm. As mentioned, the first step is to identify the centroid of an image C_x, C_y, which is often not a whole number. Given that in the first rotation the image is binary in nature, the centroid can be simply found by calculating the average coordinates of all the pixels that are bright (brightness value of 255). This is described in Equation 2.2.

$$C_x = \frac{1}{N}\sum^{N} x \quad \text{if} \quad I(x,y) = 255 \tag{2.2a}$$

$$C_y = \frac{1}{N}\sum^{N} y \quad \text{if} \quad I(x,y) = 255 \tag{2.2b}$$

Following the calculation of the centroid for each image, the next step is to find the coordinate for backward mapping of rotation coordinates for interpolation. To explain the need for brightness interpolation, we first recognise that both the original and rotated image are displayed in a discrete raster for any digital device. Figure 2.7 shows how an image, when rotated, superimposes over the discrete raster which will form the final rotated image. After rotating, the image coordinates are almost never a whole number. This is obvious in the Figure as the blue dots (representing the roatated pixel coordinates) don't align with the red dots (the discrete raster coordinates).

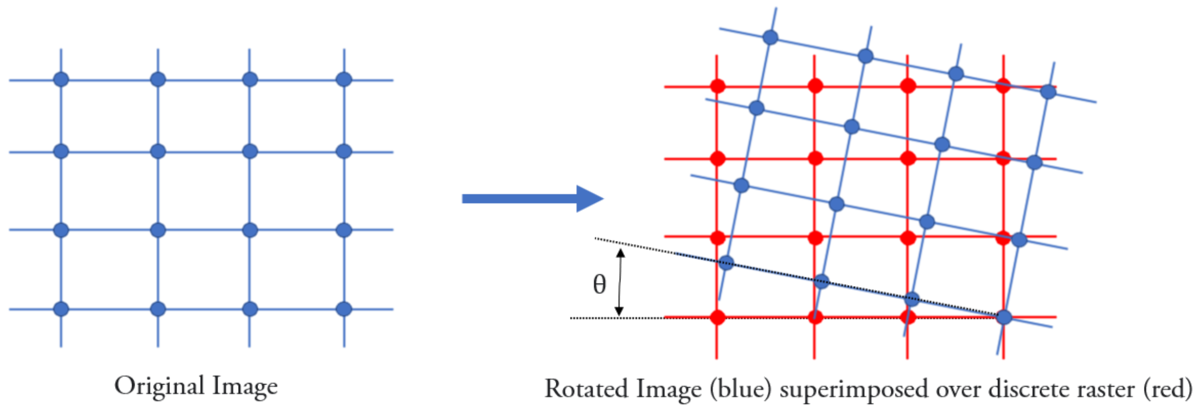

Original Image
θ
Rotated Image (blue) superimposed over discrete raster (red)

FIGURE 2.7: Rotated image superimposed over a discrete raster. The Blue and Red dots refer to the pixel location.

In order to ensure that there is a legitimate result for pixel values of the discrete raster, a backward mapping of the final image raster coordinates is done. This means to say that the discrete raster, which represent the coordinates for the rotated final image, is rotated in the opposite direction and superimposed over the original image. This is visualised in Figure 2.8. Then an interpolation is conducted, where the brightness values of the original image (blue dots), surrounding a pixel coordinate in the final image discrete raster (red dot) is interpolated to find the brightness value of the red dot. To determine the coordinates of the red points when rotating by angle θ counter-clockwise is shown in Equation 2.3, where x' and y' refer to the rotated coordinates.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.3}$$

There are three main types of interpolations widely used: Nearest Neighbour, Bilinear & Bicubic. These types can be visualised in Figure 2.9, which show the influence that surrounding pixel values in the original image have over the discrete final image raster pixel. Nearest Neighbour interpolation is the simplest, as it simply takes the brightness value of the pixel nearest to it. In bilinear interpolation, the brightness of the red dot is determined by the surrounding 4 pixels. Given that the horizontal and vertical distances from the bottom left blue dot to the red dot are a and b respectively and $g(.)$ refers to the brightness of the original image (blue dot) and $f(x, y)$ is the interpolated brightness of red dot (at location x, y), their relationship for bilinear interpolation is shown in Equation 2.4 where $l = ceil(x), k = ceil(y)$.

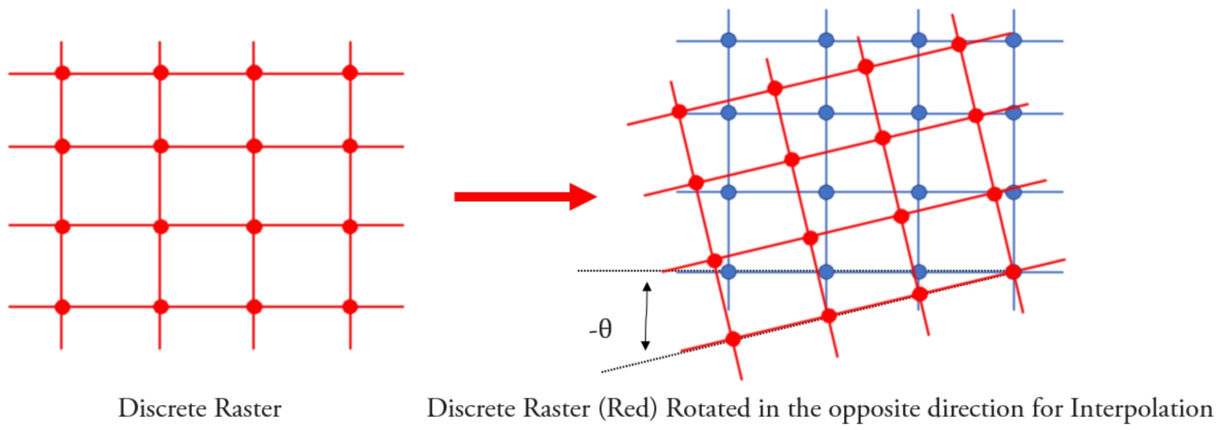

-θ
Discrete Raster
Discrete Raster (Red) Rotated in the opposite direction for Interpolation

FIGURE 2.8: Backward mapping of final image raster into the original image for interpolation

$$f(x,y) = (1-a)(1-b)g(l,k) + a(1-b)g(l+1,k) + b(1-a)g(l,k+1) + abg(l+1,k+1) \tag{2.4}$$

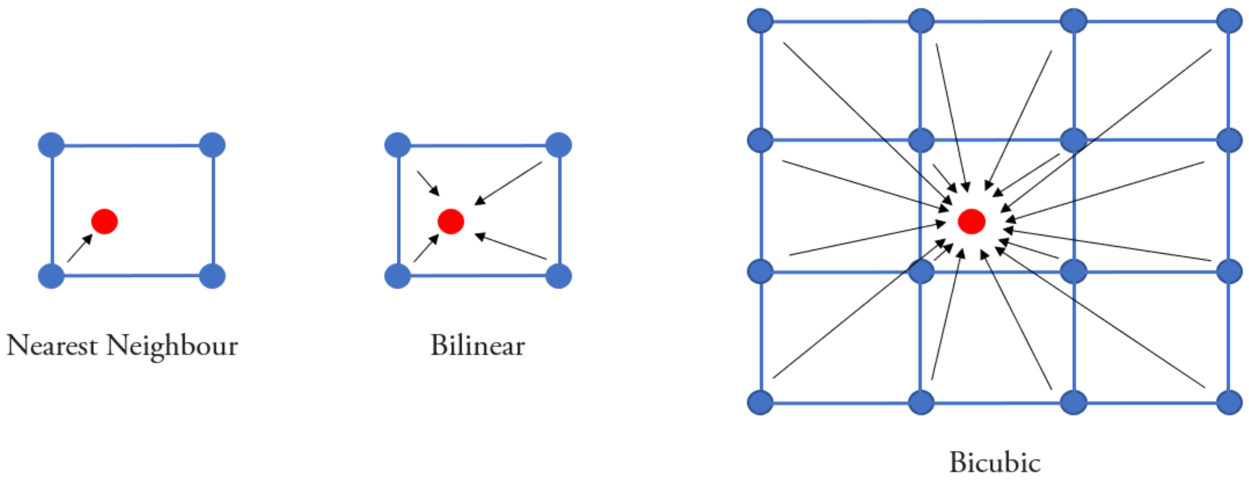


FIGURE 2.9: Types of Interpolation from blue dots into red dot

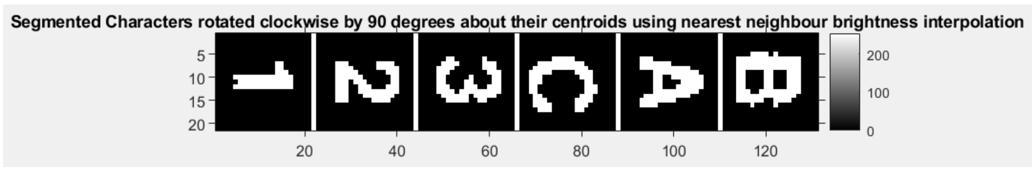
Figure 2.9 also shows the third type of brightness interpolation, bicubic interpolation, which is known to be the most accurate and common form of interpolation used in state-of-the-art softwares that deal with images. The surrounding 16 blue dots are used for interpolation into the red dot. The relation for this can be found in Equation 2.5.

$$h(\zeta) = \begin{cases} 1 - 2|\zeta|^2 + |\zeta|^3, & 0 \leq |\zeta| < 1 \\ 4 - 8|\zeta| + 5|\zeta|^2 - |\zeta|^3, & 1 \leq |\zeta| < 2 \end{cases} \tag{2.5}$$

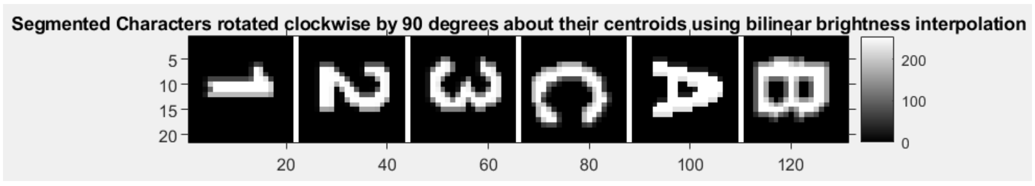
$$f(x, y) = \sum h(x)h(y)g(.)$$

2.4.1 90° Clockwise

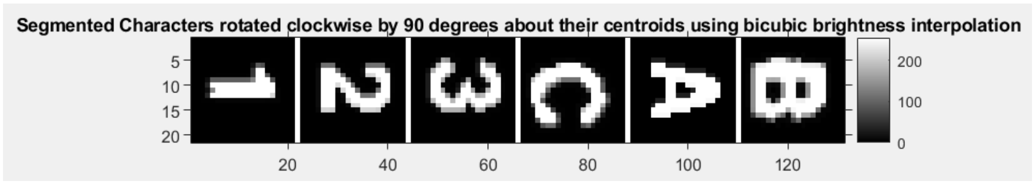
The first rotation task required us to rotate each one of the segmented character images 90° clockwise. Figure 2.10 shows the outcome of rotating the characters of Image 1 about their centroids using all three interpolation methods. Interestingly, it seems as though the nearest neighbour interpolation gave the clearest image as compared to the other two. Given that the images are still binary, a high contrast is maintained. The bilinear and bicubic interpolation methods appeared to have a smoothing and blurring effect along the edges of the characters. This is expected as the bilinear and bicubic interpolation equations tell us that there will be some pixels that are neither 0 nor 255 in brightness value, unlike the case in nearest neighbour. As the rotation is just 90°, there is no need for any extensive smoothening as the pixels can be easily mapped by switching indices, similar to how the image inversion for Image 2 was done. Another difference is that the bilinear interpolation in general has a lower brightness value along the thickness of the characters as compared to bicubic, leading to a lower contrast in Figure 2.10b. This may be due to the manner in which bicubic interpolation changes the brightness with distance, which is a smooth curve. On the other hand, the bilinear is a linear drop with distance, which could signal that the edges of the characters have a more obvious looking change with little pixels that have high brightness values. These observations are further emphasised due to the general low resolution of the image itself. These changes along the edge would become a lot less apparent with more pixels representing the thickness of the characters.


Segmented Characters rotated clockwise by 90 degrees about their centroids using nearest neighbour brightness interpolation

(A) Nearest Neighbour Interpolation


Segmented Characters rotated clockwise by 90 degrees about their centroids using bilinear brightness interpolation

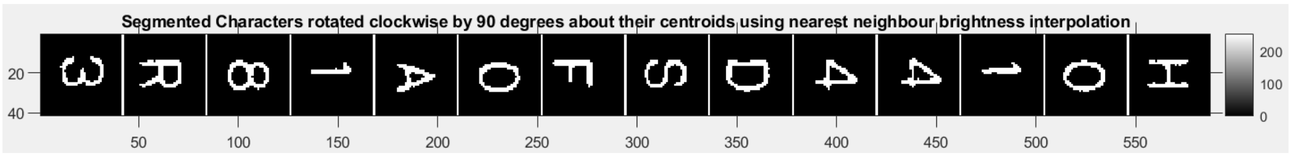
(B) Bilinear Interpolation


Segmented Characters rotated clockwise by 90 degrees about their centroids using bicubic brightness interpolation

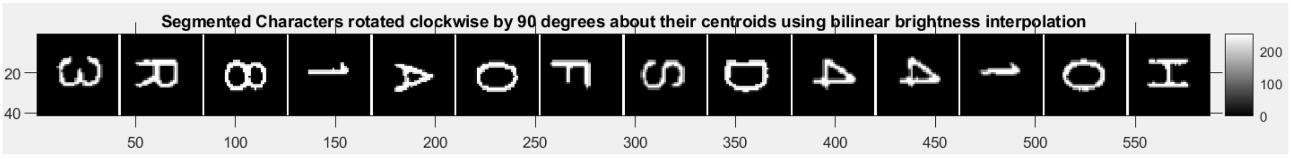
(C) Bicubic Interpolation

FIGURE 2.10: Segmented Images of Image 1 after 90° clockwise rotation about centroid

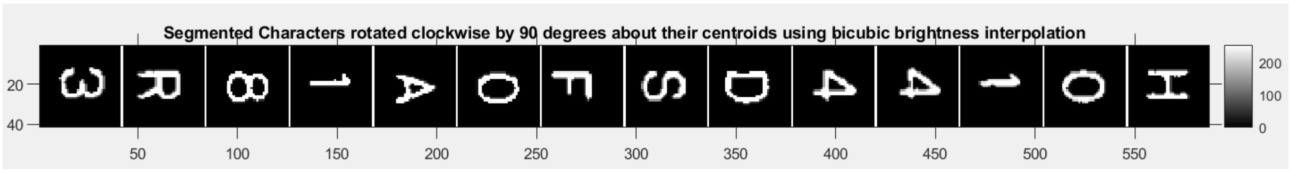
In the case for Image 2 however, Figure 2.11 shows that the difference between the three interpolation methods is not visibly very different. There is still evidence of smoothening of the edges along the bilinear and bicubic interpolated images. In the case for bicubic interpolation, the characters in the images appear to have a more consistent thickness along the characters, which the original images and nearest neighbour interpolated images do not show. There are also some signs of the images from bilinear interpolation having lower brightness as compared to the other two methods.


Segmented Characters rotated clockwise by 90 degrees about their centroids using nearest neighbour brightness interpolation

(A) Nearest Neighbour Interpolation


Segmented Characters rotated clockwise by 90 degrees about their centroids using bilinear brightness interpolation

(B) Bilinear Interpolation

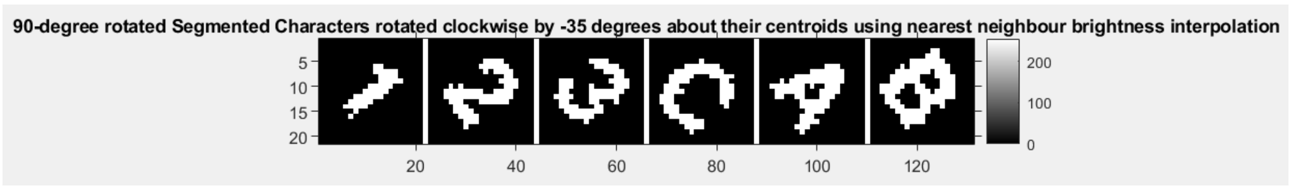

Segmented Characters rotated clockwise by 90 degrees about their centroids using bicubic brightness interpolation

(C) Bicubic Interpolation

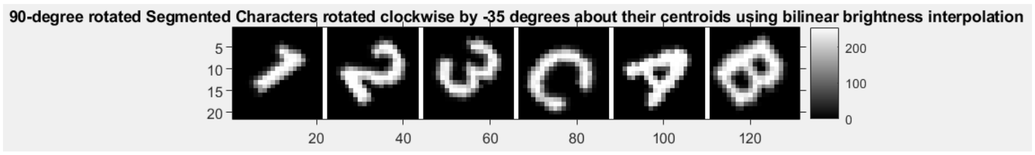
FIGURE 2.11: Segmented Images of Image 2 after 90° clockwise rotation about centroid

The next task is to take the new images that were rotated and locate new centroids for them, which is used as the new pivot for a 35° counter clockwise rotation. The results of Figure 2.12 finally show the real differences between the three interpolation algorithms in Image 1. Figure 2.12a clearly shows that the nearest neighbout algorithm performs very poorly when the the angle of rotation is not a multiple of a right angle. A slant angle reveals the weakness in the ability for nearest neighbour to accurately retain information of the image brightness. Even though the contrast is maintained, given that the image is still binary, the new edges along the initially-straight edges have become very jagged, almost staircase-like. The irregular edges are expected as the pixels are unable to take on any brightness value other than 0 or 255. For a slant image like this, a certain degree of smoothening is require to retain the morphological characteristics of the characters. This is provided by both bilinear and bicubic interpolation methods. Bilinear interpolation shows a degree of smoothening around the edges. However, this smoothening and blurring extends throughout the body of the characters, leading to a lower brightness and therefore lower contrast. The bicubic interpolation on the other hand, has very good smoothening around the edges, while maintaining high brightness in the body of the characters and hence also maintaining high contrast. One side effect of bicubic interpolation is the morphological dilation of the characters. This is visibly apparent in the letters 'A' and 'B' in Figure 2.12c, where the originally black gaps inside the characters have gotten a lot smaller.

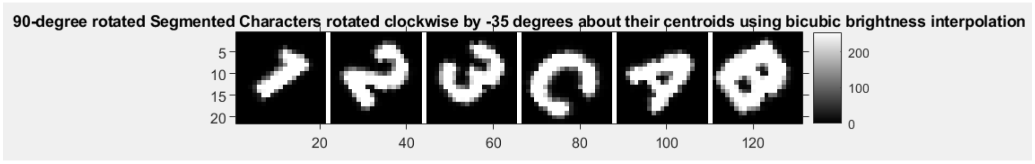
2.4.2 35° Counter-Clockwise



(A) Nearest Neighbour Interpolation



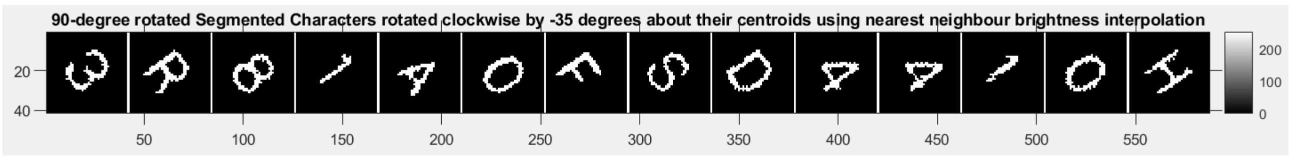
(B) Bilinear Interpolation



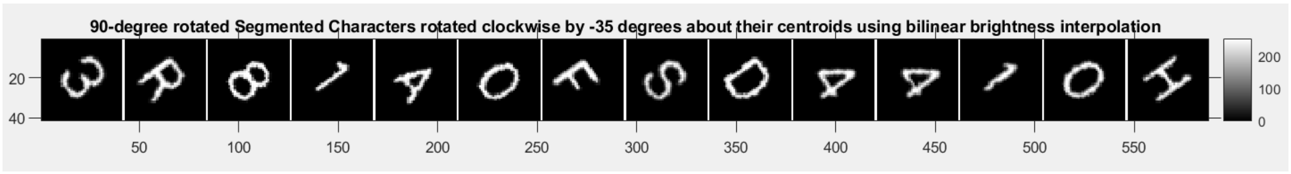
(C) Bicubic Interpolation

FIGURE 2.12: Segmented Images of Image 1 after 35° clockwise rotation about centroid (after having rotated 90° clockwise)

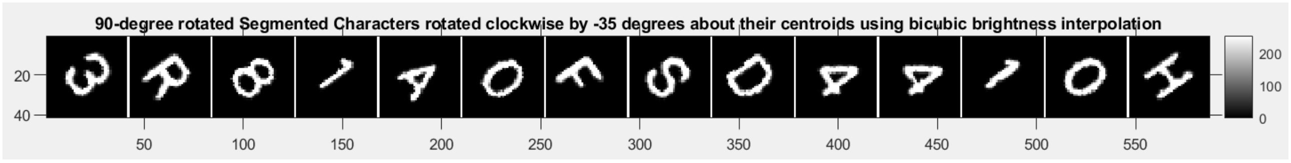
Finally, we have the exact same observations for Image 2 in Figure 2.13 that we had for Image 1. The images from nearest neighbour interpolation show irregular and jagged lines along the edges of the characters, while images from bilinear and bicubic interpolation do not show the same issue, with their smoothening along the edges. The images from bilinear interpolation also show a lower level of contrast and brightness as was seen in Image 1 (Figure 2.12b). The images from bicubic interpolation does not have this problem, and it's smoothening has a morphological dilation effect which effectively increases the thickness of all the characters, as seen in Figure 2.13c. In general, it appears that for image rotation, bicubic interpolation, retains the best information, especially for slant angle rotations. The nearest neighbour interpolation only accurately retains image information in rotations for multiples of 90°. The bilinear interpolation appears to be a compromise between the other two methods, but it has a problem of low contrast in areas of an image that are thin, like that of the low resolution images processed in this project.


90-degree rotated Segmented Characters rotated clockwise by -35 degrees about their centroids using nearest neighbour brightness interpolation
20
40
50
100
150
200
250
300
350
400
450
500
550
200
100
0

(A) Nearest Neighbour Interpolation


90-degree rotated Segmented Characters rotated clockwise by -35 degrees about their centroids using bilinear brightness interpolation
20
40
50
100
150
200
250
300
350
400
450
500
550
200
100
0

(B) Bilinear Interpolation


90-degree rotated Segmented Characters rotated clockwise by -35 degrees about their centroids using bicubic brightness interpolation
20
40
50
100
150
200
250
300
350
400
450
500
550
200
100
0

(C) Bicubic Interpolation

FIGURE 2.13: Segmented Images of Image 2 after 35° clockwise rotation about centroid (after having rotated 90° clockwise)

2.5 Outline of Images

The next task for this project is to take each one of the segmented images, and extract the outline of the images. In particular, the current project implements 3 different approaches to this step, and analyses which performs the best. The first approach is to use high-pass filtering in the spatial frequency domain. The second approach is to use high-pass filtering again, but in the spatial domain instead. The final approach is to use a mathematical morphological approach of erosion to extract the outline of the images.

2.5.1 Spatial Frequency Domain

An image is most commonly seen in the spatial domain, where the values in an array correspond to the brightness of the pixels and the location of each pixel corresponds to that of the image being seen to the observer. Another approach to seeing a picture is that of the spatial **frequency** domain. This involves a conversion as shown in Equation 2.6, describing the 2D discrete Fourier transform. $F(u, v)$ represents the frequency domain while X and Y refer to the size of the image in the horizontal and vertical directions, with x and y referring to the coordinates of the image.

$$F(u,v) = \frac{1}{XY}\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} f(x,y)e^{-i2\pi(\frac{ux}{X}+\frac{vy}{Y})} \tag{2.6}$$

The high frequency regions of the transform usually correspond to areas in the image that have more rapid changes in brightness, which will often align with the edges of an image body. The low frequency region refers to otherwise. In this project, as we aim to retain the edges only, a high pass filtering has to be done in the spatial frequency domain. There are two methods that we will explore in particular: Butterworth filtering [3] and Gaussian filtering. Equations 2.7a and 2.7b show the transfer functions of Butterworth and Gaussian high-pass filtering respectively. Note that $D(u, v)$ refers to the distance from the point (u, v) to the centre of the frequency of the rectangle, with the centre of the rectangle being zero-frequency centered. D_0 (and n for Butterworth) are user specified parameters that can be tuned. The resulting $H(u, v)$ is the resulting filtered 2D frequency domain function.

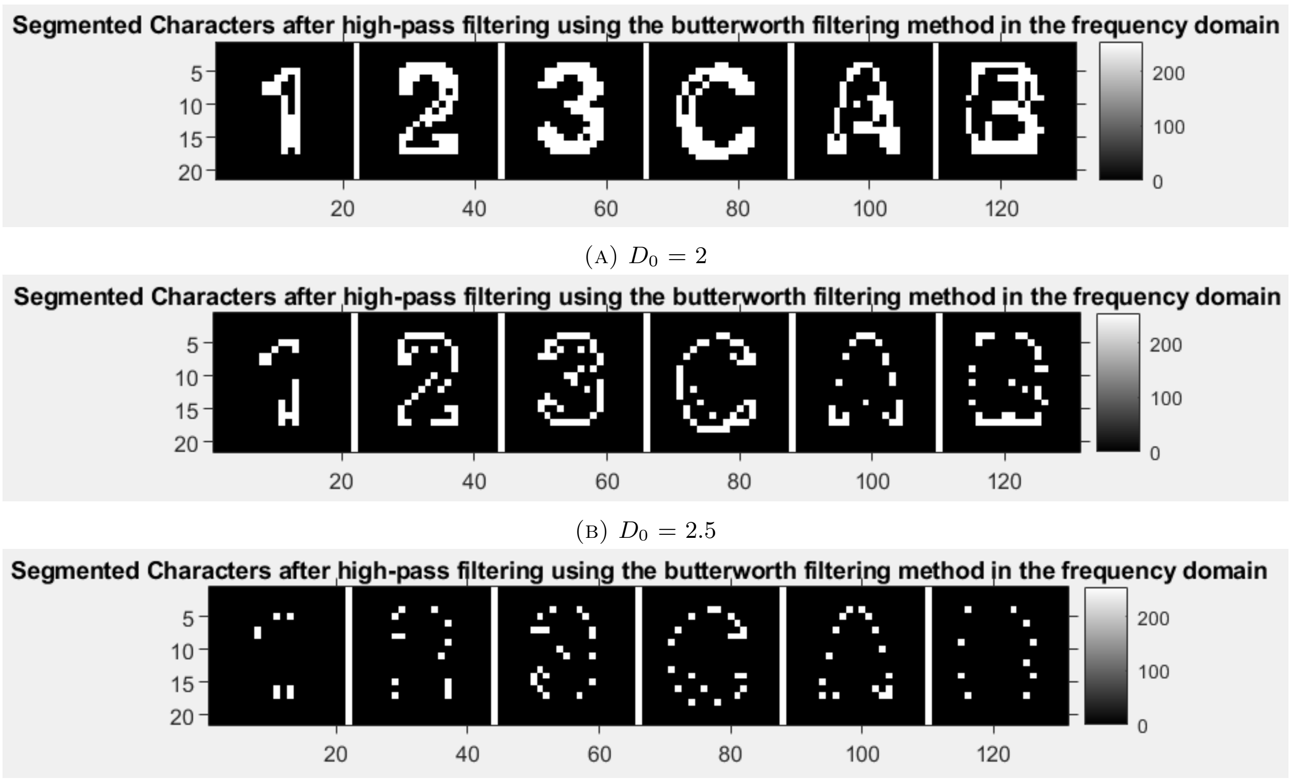
$$H(u,v) = 1 - \frac{1}{1 + [D(u,v)/D_0]^{2n}} \tag{2.7a}$$

$$H(u,v) = 1 - e^{-D^2(u,v)/2D_0^2} \tag{2.7b}$$

Following the filtering operation, the 2D inverse discrete Fourier transform as shown in Equation 2.8 is applied to restore the image into spatial domain.

$$f(x,y) = \sum_{u=0}^{X-1}\sum_{v=0}^{Y-1} F(u,v)e^{i2\pi(\frac{ux}{X}+\frac{vy}{Y})} \tag{2.8}$$

First we analyse the effect of Butterworth filtering in Image 1. As both D_0 and n affect each other in the filtering, the value of n is kept constant at 1 while D_0 is varied. Figure 2.14 shows the images after filtering at three different D_0 values. In the case where D_0 is equal to 2 in Figure 2.14a, only a small portion of the non-outline body pixels are removed, hence the outlines are not made clear. In Figure 2.14c, while all the non-outline body pixels are removed, majority of the outline is eroded away as well. Hence, the best compromise comes at $D_0 = 2.5$ in Figure 2.14b, which achieves the best balance of both retaining the outline and filtering most of the non-outline body pixels. However, the filtering at this stage is still unsatisfactory given the numerous discontinuities in the outlines.



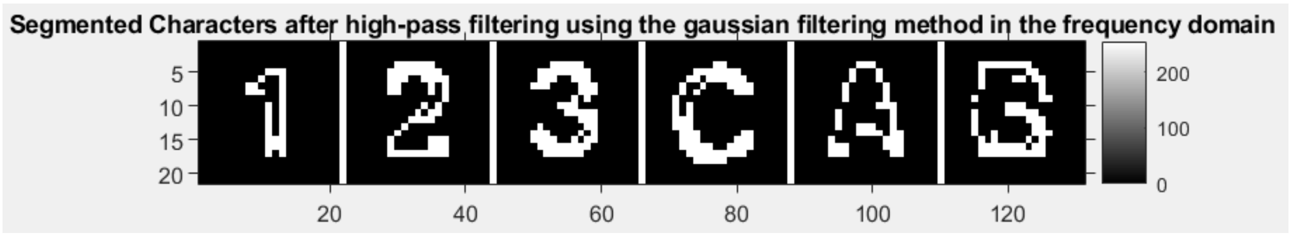
(A) $D_0 = 2$

(B) $D_0 = 2.5$

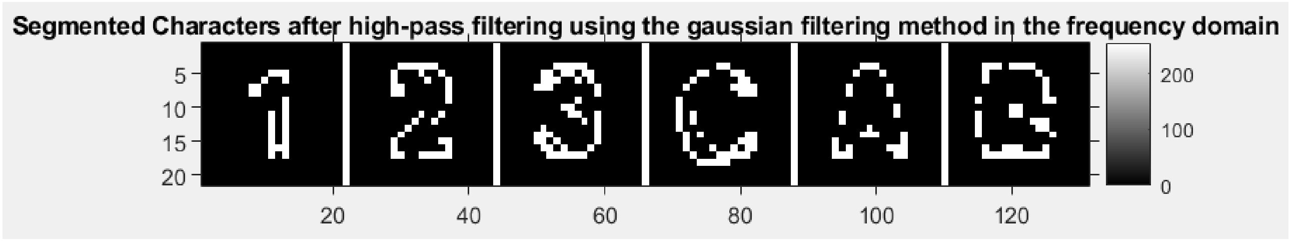
(C) $D_0 = 3$

FIGURE 2.14: Segmented Images of Image 1 after high pass filtering using the Butterworth filter to extract image outline

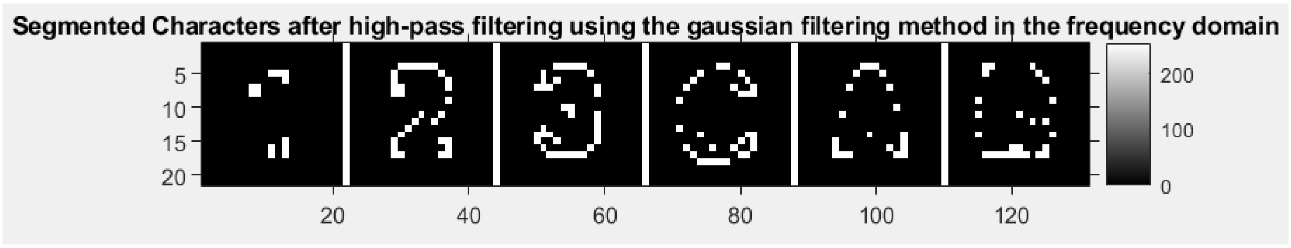
In the case for Gaussian filtering, we also varied the D_0 value. The outcome was similar to that in Butterworth filtering, where a D_0 value of 1.9 did not have a strong enough filtering effect as seen in Figure 2.15a. The D_0 of 2.4 in Figure 2.15c overshot the balance as it eroded away most of the outline, similar to Figure 2.14c. The best balance was found at $D_0 = 2.15$ in Figure 2.15b, although the outlines are also highly unsatisfactory given the numerous discontinuities in the outlines.



(A) $D_0 = 1.9$



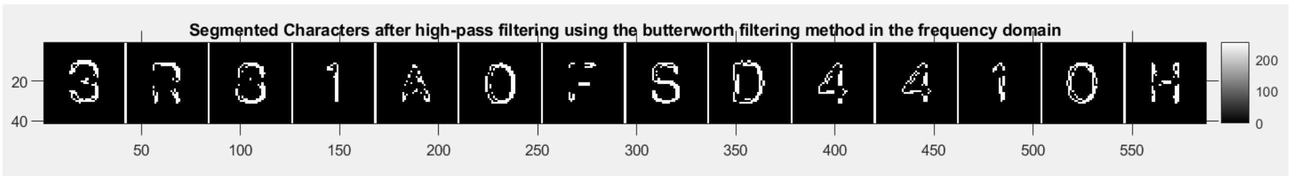
(B) $D_0 = 2.15$



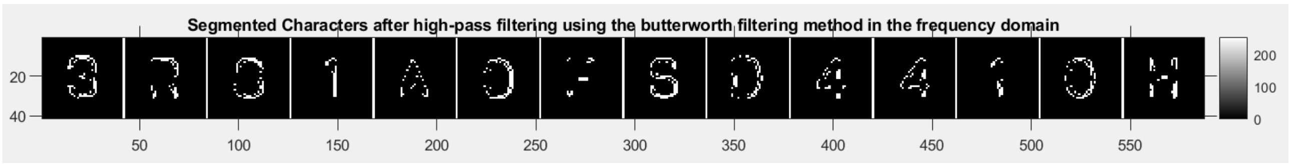
(C) $D_0 = 2.4$

FIGURE 2.15: Segmented Images of Image 1 after high pass filtering using the Gaussian filter to extract image outline

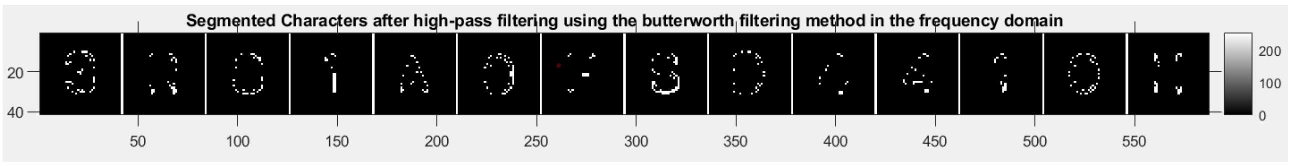
Moving on to Image 2, we tuned the D_0 values using both Butterworth and Gaussian filtering as seen in Figures 2.16 and 2.17. In both cases, the D_0 value of 4.5 achieved the best balance between filtering away the body pixels and retaining the outline of the characters. However, just like Image 1, none of the results are satisfactory given the numerous discontinuities in the outlines. There could be numerous reasons for this, one being that the images are inherently very low in resolution. As a result, the frequency domain may not be accurate enough with so few pixels. In particular, having very few pixels to represent the thickness of the body of the characters make it very challenging for any filter to clearly filter out the non-outline pixels. Nonetheless, the results from the spatial frequency domain has shown little competence in handling this task of the project. Hence, we move on to filtering in the spatial domain.


Segmented Characters after high-pass filtering using the butterworth filtering method in the frequency domain

(A) $D_0 = 4$

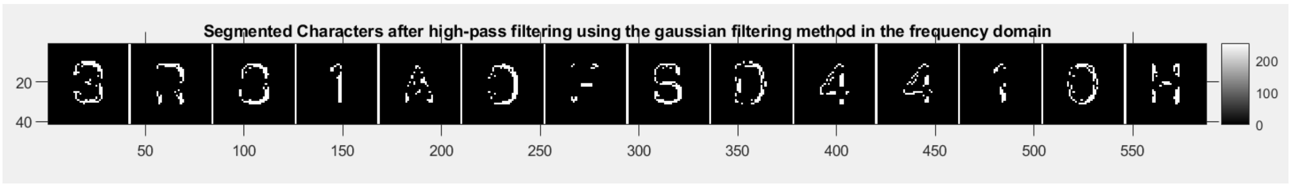

Segmented Characters after high-pass filtering using the butterworth filtering method in the frequency domain

(B) $D_0 = 4.5$

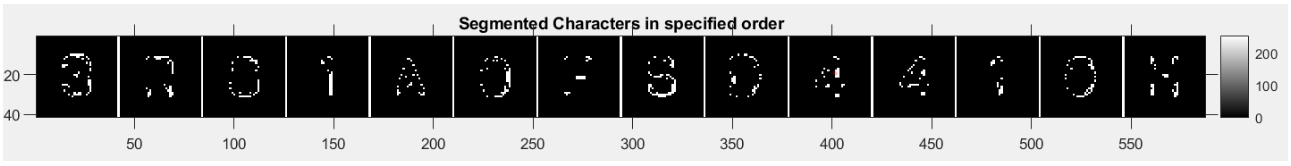

Segmented Characters after high-pass filtering using the butterworth filtering method in the frequency domain

(C) $D_0 = 5$

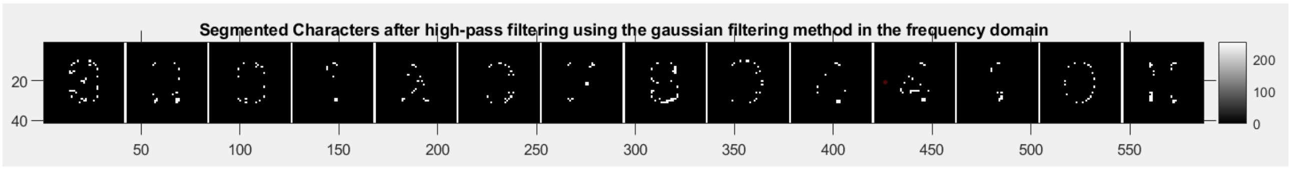
FIGURE 2.16: Segmented Images of Image 2 after high pass filtering using the Butterworth filter to extract image outline


Segmented Characters after high-pass filtering using the gaussian filtering method in the frequency domain

(A) $D_0 = 4$



(B) $D_0 = 4.5$


Segmented Characters after high-pass filtering using the gaussian filtering method in the frequency domain

(C) $D_0 = 5$

FIGURE 2.17: Segmented Images of Image 2 after high pass filtering using the Gaussian filter to extract image outline

2.5.2 Spatial Domain

Filtering in the spatial domain does not require the use of Fourier transform. It utilises a mask that is applied across the image using convolution with the brightness values of the pixels. This is shown in Figure 2.18, where the value of one cell is determine as a linear combination of the weights in the mask and the brightness values of the image. This mask is applied with the centre of the mask being placed on every pixel in the image, to determine the new value of the pixel, a process known as local processing. If any one of the cells of the mask fall outside the image, like that in the outer boundary of pixels in an image, those weights are simply multiplied by zero.

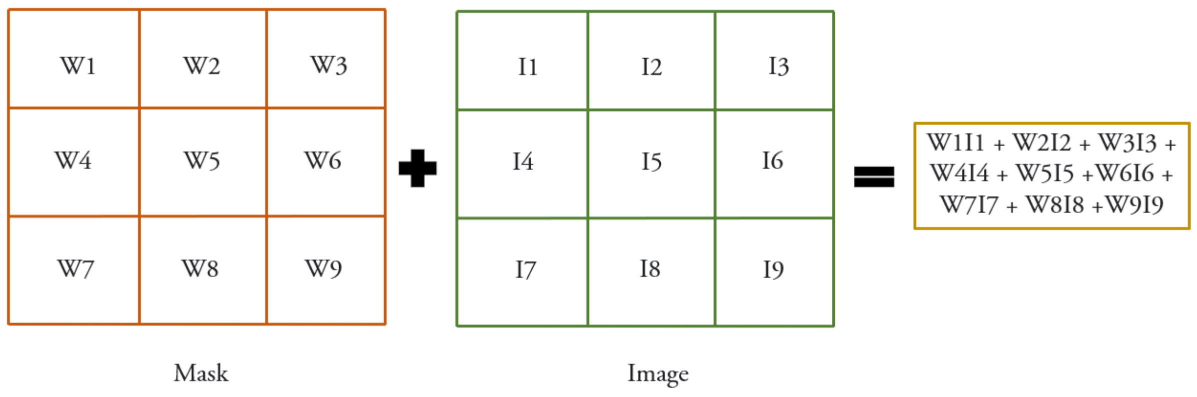


FIGURE 2.18: Example of convolution of a mask on an image

There are well-known masks that have the ability to conduct edge detection in images. One such example is the Prewitt operator [4] that uses the gradient change along an image to identify edges. P_x and P_y in Equation 2.9 are a pair of masks that are applied throughout the Image, I. The P_x mask is used to identify vertical edges while P_y is used to identify horizontal edges. The final value in the pixel is determined as the magnitude $P = \sqrt{P_x^2 + P_y^2}$.

$$P_x = \begin{pmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{pmatrix} * I \quad \text{and} \quad P_y = \begin{pmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} * I \tag{2.9}$$

Another operator that is also popular in edge detection is the Sobel operator [5]. It looks similar to the Prewitt operator, as shown in Equation 2.10. Both the Prewitt and Sobel operators use the idea of gradient change in image brightness for edge detection. S_x and S_y are used to detect edges in the vertical and horizontal directions as both masks are applied throughout the image. Just like the Prewitt operator, the final value of the pixel is calculated as $S = \sqrt{S_x^2 + S_y^2}$.

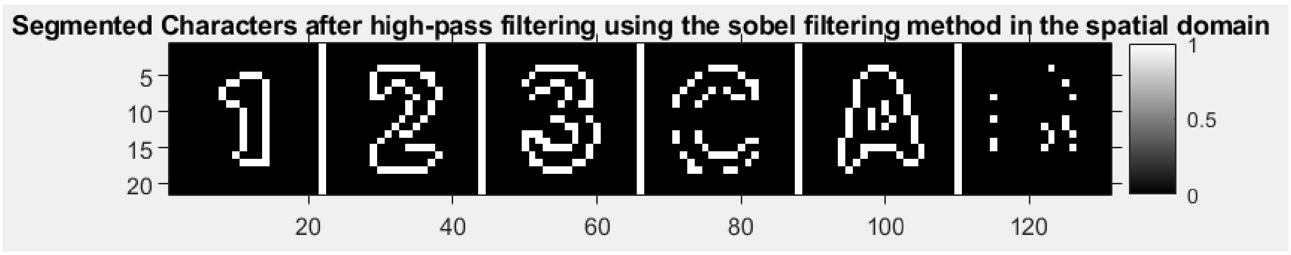
$$S_x = \begin{pmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{pmatrix} * I \quad \text{and} \quad S_y = \begin{pmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} * I \tag{2.10}$$

The final algorithm used in the spatial domain with masks is the Canny edge detection algorithm [6]. The algorithm is a lot more complex than simply applying a mask. The first stage requires an application of the Gaussian filtering with a K $\times$ K mask. The larger the size of the mask, the lower the sensitivity to noise, but the location of the edge may become less accurate. It is common to use a K of 5 for the Canny edge detection algorithm. The weights of the mask are described in i & j coordinates in Equation 2.11.

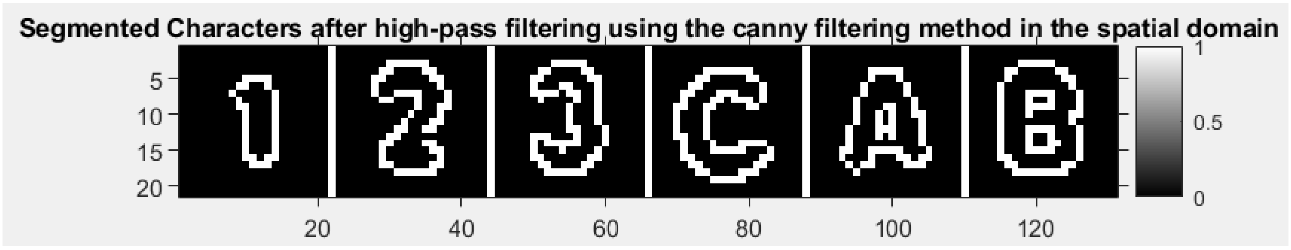
$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left\{ -\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2} \right\}, \quad 1 \le i, j \le 2k+1 \tag{2.11}$$

The second step in the Canny edge detection algorithm is to use a gradient operator like the Prewitt or Sobel operators as described in Equations 2.9 & 2.10. The third step is to conduct non-maximum suppression on the image to aid in thinning down the edges of the image. This is done by comparing the value of a pixel with the positive and negative edge directions. If the value of the current pixel is the largest compared to the pixels in the same edge direction, the edge value is retained, otherwise it is suppressed. Following that, the fourth step of the algorithm uses a double threshold to further suppress more pixels that may still have a prominent value due to noise. If the pixel value is lower than the specified low threshold, it is suppressed. On the other hand, if the pixel value is higher than the specified high threshold, it is marked as a strong edge. Any other pixels that fall between the low and high thresholds are marked as weak edges. To resolve the weak edges, if it is connected to a strong edge pixel via the 8-connectivity described in Figure 2.5, it is assumed to be a true edge and hence marked as a strong edge; otherwise, the weak edge is suppressed. After all these steps are over, the strong edge pixels are then treated as the final edge pixels. For this project, the Canny edge detection algorithm is handled entirely by MATLAB's built-in edge function.

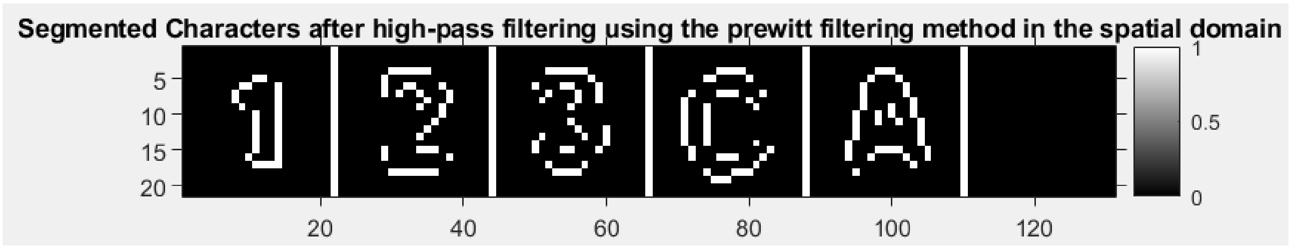
Figure 2.19 shows the results of using all the three spatial methods to detect the outline of the characters in Image 1. The first key observation is that all three operations perform significantly better than the entirety of high-pass filtering in the spatial frequency domain. Generally, all three edge detection operations managed to satisfactorily get the outline of the image, with the exception of the letter 'B'. In Figure 2.19c, the letter 'B' has been completely filtered out while in Figure 2.19a, there are only a few pixels of the remnants of the letter 'B'. We are not completely sure why this is the case, however the Canny edge detection algorithm in Figure 2.19b had no issues in generating the outline for the letter 'B' perfectly. In fact, the Prewitt and Sobel operators did not managed to trace the outlines perfectly as there are still some places along the outlines where there are gaps. The gaps are more obvious for the images after filtering with the Prewitt operator. In that manner, it is clear that the Canny edge detection worked perfectly on all the characters for Image 1.


Segmented Characters after high-pass filtering using the sobel filtering method in the spatial domain
5
10
15
20
20
40
60
80
100
120
1
0.5
0

(A) Sobel Operator Filtering


Segmented Characters after high-pass filtering using the canny filtering method in the spatial domain
5
10
15
20
20
40
60
80
100
120
1
0.5
0

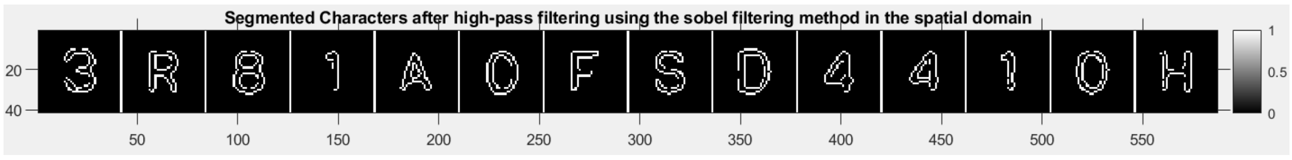
(B) Canny Edge Detection


Segmented Characters after high-pass filtering using the prewitt filtering method in the spatial domain
5
10
15
20
20
40
60
80
100
120
1
0.5
0

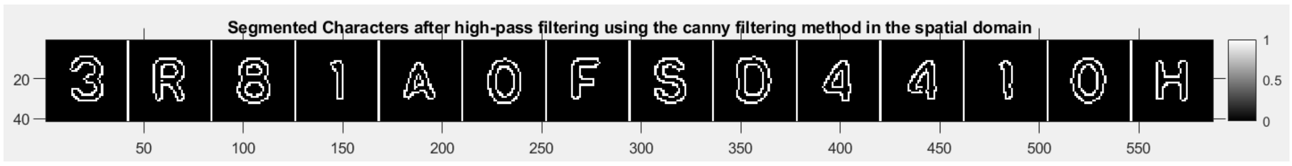
(C) Prewitt Operator Filtering

FIGURE 2.19: Segmented Images of Image 1 after high pass filtering in the spatial domain

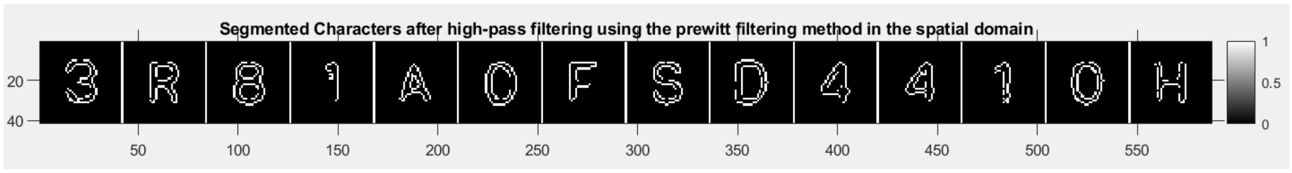
The results for Image 2 are very similar to that of Image 1. In Figure 2.20, we see that all the three methods performed satisfactorily on the characters in each image. The Sobel and Prewitt operators again led to some images having gaps in the outline. The Canny edge detection worked perfectly on all the characters. It is clear from the observations that the technique of using masks in the spatial domain worked much more effectively than the filtering in the spatial frequency domain. We believe this difference may be the results of the former methods being more effective at low resolutions than the latter methods. It is possible that withe high resolution images with many pixels to represent an image body, methods from both domains will be similarly effective.


Segmented Characters after high-pass filtering using the sobel filtering method in the spatial domain

(A) Sobel Operator Filtering


Segmented Characters after high-pass filtering using the canny filtering method in the spatial domain

(B) Canny Edge Detection


Segmented Characters after high-pass filtering using the prewitt filtering method in the spatial domain

(C) Prewitt Operator Filtering

FIGURE 2.20: Segmented Images of Image 2 after high pass filtering in the spatial domain

2.5.3 Erosion Technique

The final technique in exploring the methods to extract the outline form the segmented characters is using mathematical morphology. In particular, erosion is utilised to aid the extraction of the edges. Erosion is done by taking a structural element to probe around the entire image. The formula to get the image with the outline is shown in Equation 2.12. I is the original image, S is the structured element, I^* is the final image and $\ominus$ is the erosion operator. This is also known as the internal gradient of the image. Figure 2.21 shows the process in which this edge detection occurs. A 2 by 2 square structured element, $S = \begin{pmatrix} 1^* & 1 \\ 1 & 1 \end{pmatrix}$ was used in the erosion process, where the $*$ refers to the location of the origin and 1 means that it is used in the morphological operation. The eroded image in Figure 2.21b is thinner than the actual image in Figure 2.21a. Hence subtracting the former from the latter will give an image like that in Figure 2.21c.

$$I^* = I - (I \ominus S) \tag{2.12}$$

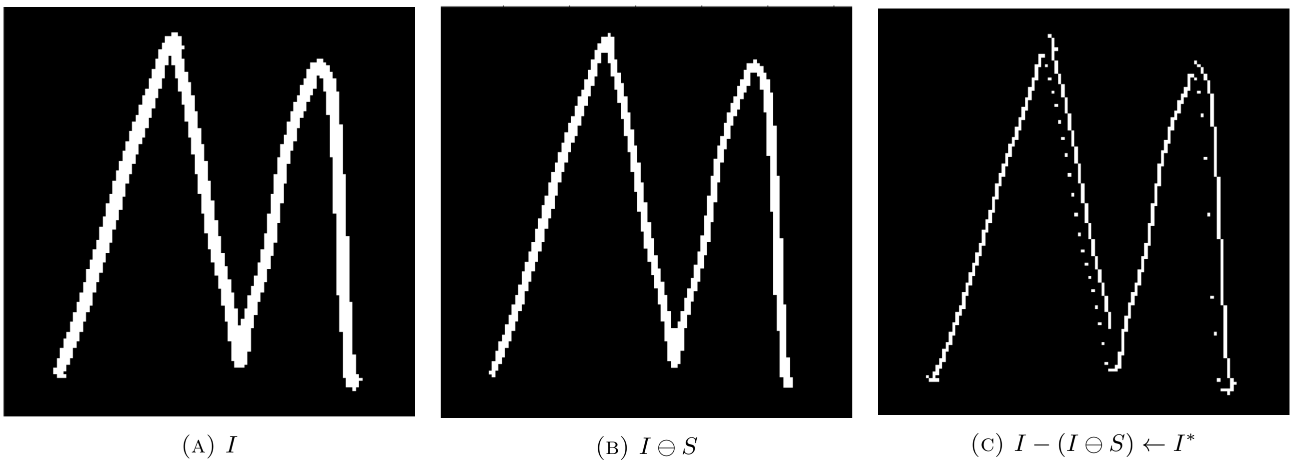
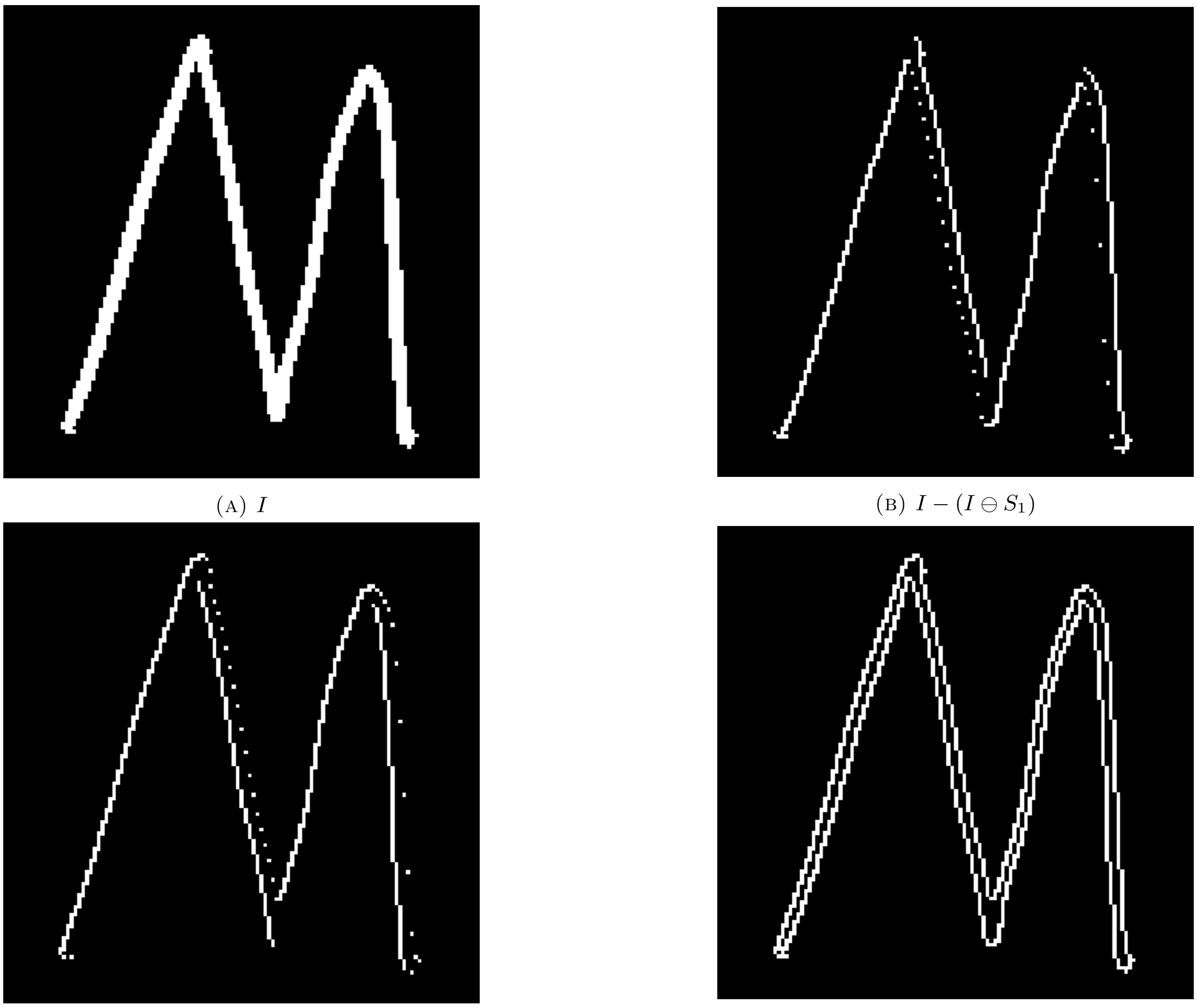

(A) I (B) $I \ominus S$ (C) $I - (I \ominus S) \leftarrow I^*$

Figure 2.21: Process of getting internal gradient via erosion in mathematical morphology for edge detection. A 2 by 2 square structured element was used.

The edge detection here is not perfect, possibly due to the fact that the image body is very thin, hence even a 2 by 2 structured element may be too large. A 3 by 3 structured element, given its large size, would erode most the image as only a minority of the pixels can support the structured element. As a result, when subtracting the image with the post-erosion image, most of the image would be retained, hence being not effective. On the other hand, a smaller 1 by 1 structured would not make sense as it would match every single image body pixel, retaining the entire image post erosion. As a result, upon subtracting the image with the post-erosion image, the entire image would be eroded away. However we did notice that for some reason the image in Figure 2.21c has perfect edges on the right side of any part of the image body. It made us realise that perhaps there is a way to extract edges on the left side in a similar manner, and sum it to the image in Figure 2.21c. We suspected that this was a result of the origin of S being at the top left. What if we made the origin at the bottom right instead? As a result, we redefined Equation 2.12 into a new double-element edge detection method as described in Equation 2.13. We would like to point out that this was of our own finding and implementation and we did not read this in any other sources.

$$I^* = [I - (I \ominus S_1)] + [I - (I \ominus S_2)], \quad S_1 = \begin{pmatrix} 1^* & 1 \\ 1 & 1 \end{pmatrix} \quad \text{and} \quad S_2 = \begin{pmatrix} 1 & 1 \\ 1 & 1^* \end{pmatrix} \tag{2.13}$$

With this new technique, we implemented it on the same example as in Figure 2.21. Figure 2.22 shows that our proposed method was successful. Figure 2.22c shows what we were missing, the edges on the left side of the image body. Summing this with the image in Figure 2.22b, we get a perfectly outlined image in Figure 2.22d. Given that the image objects in Image 1 and Image 2 are even more thin, we decided to adopt our proposed method on them too.



(A) I

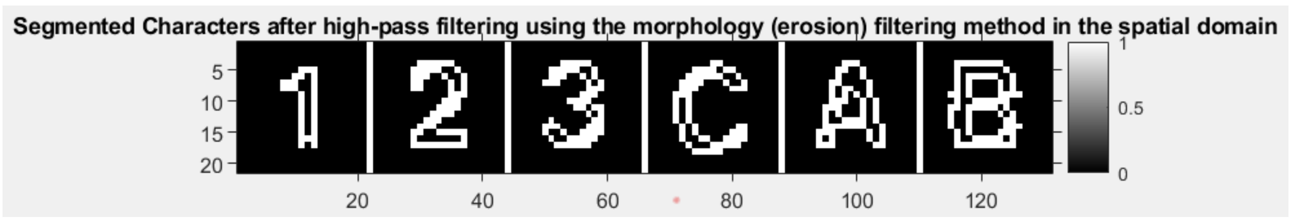
(B) $I - (I \ominus S_1)$

(C) $I - (I \ominus S_2)$

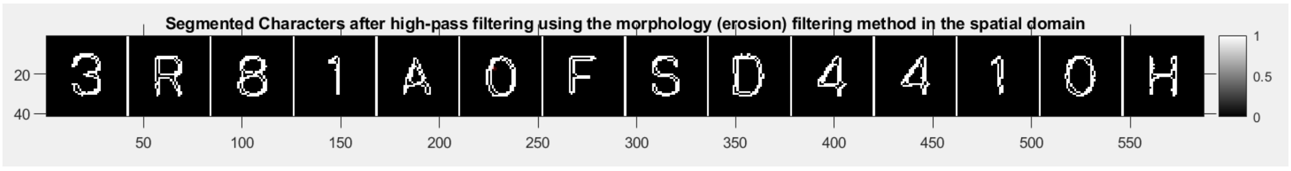
(D) $[I - (I \ominus S_1)] + [I - (I \ominus S_2)] \leftarrow I^*$

FIGURE 2.22: Process of getting internal gradient via erosion in mathematical morphology for edge detection. The double-element edge detection method described in Equation 2.13 is adopted

Figure 2.23 illustrates the results of using the proposed double-element edge detection as describer in Equation 2.13. It can be seen that the outline extraction was not perfect. In both images, it can be seen that the outline is clearly captured, however much of the image body pixels which should have been eroded still remain. This is clearly a result of the thickness of the characters being too thin. We are confident that if the image body is thicker, like that in Figure 2.22, the edge detection would be mostly perfect. To conclude all of the methods used to do outline extraction for the images in this project, the Canny edge detection algorithm performed perfectly on all the characters from both images and is the clear winner.



(A) Image 1



(B) Image 2

FIGURE 2.23: Segmented Images after the double-element edge detection method described in Equation 2.13 is used

2.6 Pixel-Thin Images

In this step, the task set in the project is to generate a pixel-thin image for all characters of both images. This is effectively creating an image that is only one pixel thick, and still represents the morphological meaning of the original image. Hence the characters should still be recognisable. This task is also known as creating skeleton representations of the image. For this project, we simply apply one algorithm to complete this task. We use the hit-and-miss algorithm in mathematical morphology for image body thinning. The algorithm requires the use of special types of structured elements, that allow for hit-and-miss operations. For this algorithm in particular, 8 different structured elements are used as described below.

$$S_1 = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad S_2 = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & 0 \end{pmatrix} \quad S_3 = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 1 & 1 \\ -1 & 0 & 1 \end{pmatrix} \quad S_4 = \begin{pmatrix} -1 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

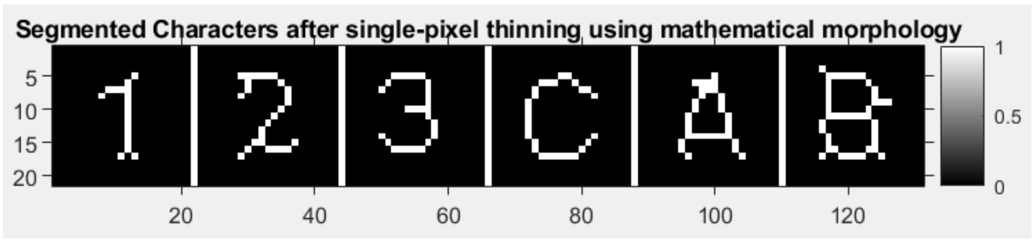
$$S_5 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ -1 & -1 & -1 \end{pmatrix} \quad S_6 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & -1 \end{pmatrix} \quad S_7 = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 1 & 1 \\ -1 & 0 & 1 \end{pmatrix} \quad S_5 = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 1 & 1 \\ -1 & -1 & 0 \end{pmatrix}$$

A 1 in this refers to pixels a structured element must hit (must have a brightness of 255) and a -1 refers to a pixel that the structured element must miss (brightness must be 0) for the overlap to be considered a positive. In traditional structured elements, a -1 is not usually considered. The algorithm for thinning can be described by pseudo algorithm 5 as shown below.

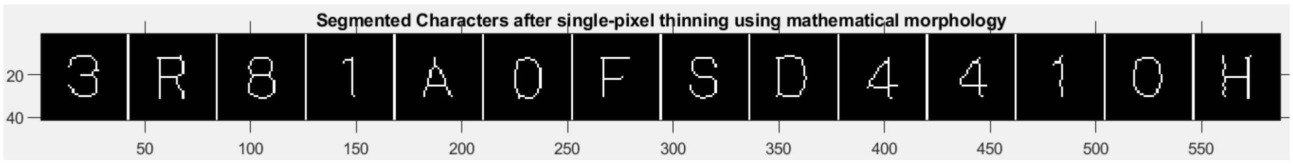
Pseudo Algorithm 5: Thinning with Hit-and-Miss Morphological Operations

```
Existing Variable(s):
    • Image array $I(x, y)$
Initialise:
    • $S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8 \leftarrow$ defined above
    • Temp Image array $I^*(x, y) \leftarrow$ zeros
Execute:
    • While $I(x, y)$ is not the same as $I^*(x, y)$:
        – $I^*(x, y) = I(x, y)$
        – $I(x, y) = I(x, y) - I(x, y) \ominus S_1$
        – $I(x, y) = I(x, y) - I(x, y) \ominus S_2$
        – $I(x, y) = I(x, y) - I(x, y) \ominus S_3$
        – $I(x, y) = I(x, y) - I(x, y) \ominus S_4$
        – $I(x, y) = I(x, y) - I(x, y) \ominus S_5$
        – $I(x, y) = I(x, y) - I(x, y) \ominus S_6$
        – $I(x, y) = I(x, y) - I(x, y) \ominus S_7$
        – $I(x, y) = I(x, y) - I(x, y) \ominus S_8$
Return $I^*(x, y)$
```

Pseudo Algorithm 5 describes the process of constantly eroding the image with the 8 hit-and-miss structured elements in the specified order until the image array converges and stops changing. When this happens, the pure skeleton form of the image is ensured to form. The application of this algorithm in the two images for this project is shown in Figure 2.24. In this case, both image 1 and image 2 have been thinned down to their skeletal form perfectly. Given that the performance of this algorithm was excellent, we saw no need to explore other algorithms for this stage.



(A) Image 1

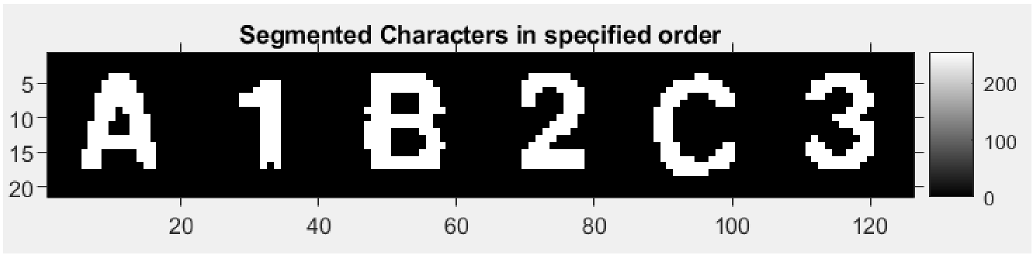


(B) Image 2

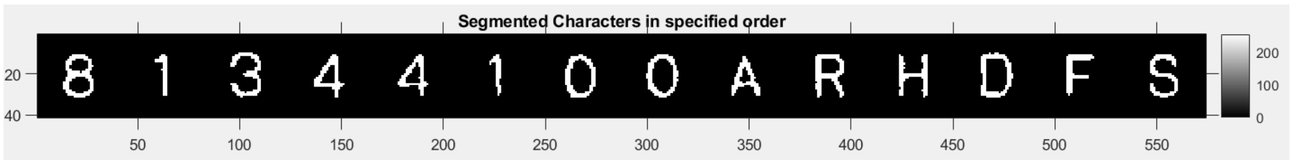
FIGURE 2.24: Pixel-thin segmented Images after thinning with the hit-and-miss algorithm in mathematical morphology

2.7 Sequence of Images

The final step of the project task is to order the characters for both images in a specified order. Specifically, we must arrange the characters with the sequence: **A1B2C3** for Image 1 and **81344100ARHDFS** for Image 2. As there was no restriction placed in conducting this step manually, we simply swapped the order of the segmented images from Figures 2.6a and 2.6b to form the specified order in Figure 2.25. This concludes the image processing workflow for this project.



(A) Image 1



(B) Image 2

FIGURE 2.25: Segmented Images ordered in the specified order for the final step

Chapter 3

Conclusion

3.1 Comparison between Image 1 & Image 2

The final aim stated in the project requirements was to review a comparison between the image processing workflow between image 1 and image 2. Even though both images undergo the exact same image processing steps and the same code was more or less used for both images, there are still some differences that we would like to highlight. One key difference we must first mention is the difference in resolution between image 1 characters and image 2 characters. The characters in image 2 use four times more pixels than in image 1. Before processing, both images had to undergo very different steps. Image 2 had to be inverted while image 1 did not, while image 1 had to map its quantization from 0-9 & A-V to 0-255 while image 2 did not. During the binary thresholding, we observed very different histograms as seen in Figure 2.3. Most of the pixels in image 1 were set to a brightness of 0, while the other pixels were spread across the rest of the spectrum. This meant that it was a lot easier to distinguish between background pixels and image object pixels. In the case for image 2, there was an obvious spectrum with two humps. As a result, both images had very different threshold values of 10 and 85 for image 1 and image 2 respectively.

During the image segmentation phase, there was virtually no difference between both images. Both images came out with perfectly segmented characters. One difference would be that the size of the character image for image 1 was 21 by 21 while it was 41 by 41 for image 2. This is expected as the character size for image 2 is on-average larger than that of image 1. The next key difference came when we attempted to rotate them by 90° clockwise. As Figure 2.10 shows, there were quite a few differences between the three brightness interpolation methods in image 1, as explained in that section. These differences were not that apparent for image 2, where the outcome looked very similar for all brightness interpolation methods. In the rotation case for 35° counter-clockwise, the irregular jaggedness for nearest neighbour interpolation was a lot more striking for image 1 as seen in Figure 2.12a, as compared to that of Image 2 in Figure 2.13a. This could mostly be attributed to the difference in number of pixels used to represent the characters between the images. As image 1 has less pixels, these jagged irregular edges become a lot more apparent to the human eye. Other than this, the only other difference worth mentioning is that in the edge detection in the spatial domain, the letter 'B' in image 1 was completely or almost filtered out by the Prewitt & Sobel operators (Figure 2.19), while no such anomaly was observed for image 2 (Figure 2.20).

Most of the aforementioned differences mentioned are very minimal and often due to small differences between the images. We found that both images worked very similarly and the image processing operations went very smoothly for both image 1 and image 2. In fact after we finished the code for one image, we could immediately reuse the same code (with minimal differences like changes in parameters) for the other image.

3.2 Further Improvements

3.2.1 Binary Thresholding

The threshold for binary thresholding was chosen simply by observing the histograms. In the case for image 1, it is relatively simply as the background was obviously demarcated as 0. However, the region between the two humps in Figure 2.3b showed that there is a relatively wide region between the two humps with no clear valley. Better improvements over these could be to use algorithms like probability thresholding and automatic thresholding.

3.2.2 Image Segmentation

From the earlier implementation of the classical algorithm, it clear drawback discussed was the space inefficiency of the classical algorithm given its memory usage in storing equivalences. Hence, a space efficient classical algorithm is developed in order to minimise the drawback of the classical algorithm. Like the classical algorithm, the initialisation step involved giving each pixel with grey-level intensity of 255 is given a unique non-zero integer label and zero otherwise. However, unlike the classical algorithm that proceeds with a top-down left-right label propagation till the bottom-right pixel, the space efficient classical algorithm label propagation proceeds top-down for only for a row of pixels, where the pixel value is replaced by the minimum value of its non-zero neighbours (based on 4-connectivity) in a recursive manner and the equivalence of the two non-zero labels from the neighbours are logged. After the necessary equivalences amongst the labels for the row of pixels in the image is generated, the union find graph search algorithm is utilised to resolve the various equivalences logged to derive the necessary equivalent classes. Lastly, a left-right label propagation is executed to relabel all the labels in the the row of pixels in the image. The stated 3 steps are repeated for each row of pixels in the image, in a top down manner. However, not all the relabelling is completed by the end of the first top-down pass, and a second bottom-up pass is required for both the remainder of the equivalence finding and for assigning the final labels. Hence, the space efficient algorithm is essentially the classical algorithm repeated per row of the image in the top-down pass as well as the bottom-up pass. As a result, the equivalences are localised to be between the current row of the image and the row that precedes it. Given that the maximum number of equivalences is the number of pixels per line, the memory required to store the equivalences is significantly reduced and the image is processed significantly faster as compared to the classical algorithm, with growing deviation in performance with increasing image size and complexity.

3.2.3 Image Rotation

Generally there were no issues with this section as we managed to successfully implement image rotation for any angle with all three types of brightness interpolation. One improvement we would make however is to see if increasing the resolution of the image would improve the brightness interpolation.

3.2.4 Image Outline

We covered this section very thoroughly, exploring a multitude of methods. One improvement we could make is to manually code out the mathematical morphological operations without relying on MATLAB's in-built imerode function, simply as a way to solidify our understanding of the algorithm and MATLAB syntax.

3.2.5 Pixel-Thin Images

We only tried one method of image thinning. Although the outcome of it was already on point, we could also try exploring with methods in mathematical morphology.

3.3 Obstacles and Learning Points

One of the key challenges faced in the project would be the initial learning curve in the coding syntax and functions utilised in MATLAB. Given that we have limited or no prior experiences with the MATLAB software, it took us some time to acclimatise to MATLAB. However, given the availability of documentation online that describes many of the in-built functions as well as the functions found in the image processing packages clearly, we were able to pick and implement the image processing tasks in good time. In particular, we realised that MATLAB is very well equipped with various image processing packages that provides countless of useful functions that are able to handle many of the image processing tasks required of in this project. While we took special effort to implement many of the functions manually in order to gain an in depth understanding on the imaging processing task, it is definitely most efficient to utilise the MATLAB image processing packages in further applications based projects.

While the image processing tasks have been fulfilled in this project, we recognise that the images used in this project are relatively simple and without any significant noise (in particular for Image 2). It is due to this fact that many of the image processing tasks have been relatively straightforward to implement and the results are in general satisfactory. Hence, this highlights the importance of having good control over the lighting conditions when capturing an image, as it would significantly simplify and make efficient the image processing steps thereafter. On the occasion where the lighting conditions are poor in addition to other environmental factors that are not properly controlled, the obtained image in reality could be of poor equality (e.g. presence of significant noise) that would result in significantly more post processing work that could otherwise be unnecessary. For example, for image enhancement against noise, spacial frequency filtering methods such as the Butterworth and Gaussian filters may be deployed. As these filters have hyperparameters that must be tuned for optimal processing, the process of hyperparameter tuning can be reasonably perceived as a tedious and cumbersome process. Hence, for any image processing task, the whole flowchart from image acquisition to image processing must be well managed from start to end for the system to be efficient and to produce optimal results.

Bibliography

[1] “Matlab optimization toolbox,” 2019.

[2] J. Hoshen and R. Kopelman, “Percolation and cluster distribution. i. cluster multiple labeling technique and critical concentration algorithm,” *Physical Review B*, vol. 14, no. 8, p. 3438, 1976.

[3] S. Butterworth *et al.*, “On the theory of filter amplifiers,” *Wireless Engineer*, vol. 7, no. 6, pp. 536–541, 1930.

[4] J. M. Prewitt, “Object enhancement and extraction,” *Picture processing and Psychopictorics*, vol. 10, no. 1, pp. 15–19, 1970.

[5] I. Sobel and G. Feldman, “A 3x3 isotropic gradient operator for image processing,” *a talk at the Stanford Artificial Project in*, pp. 271–272, 1968.

[6] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.