ME 5405
Machine Vision Project Report

Khakimjon Saidov (A0191623H)
Utkarsh Anand (A0147589J)

Introduction

In the last 4 decades, machine vision has had an important role in industry, security, medicine, and academia. Humans vision is very diverse, for example we can distinguish between different objects and can isolate the background from the object of interest very efficiently. But what human vision is not very good at is performing the same task repetitively as we tend to make more mistakes and lose neutrality as we do the same thing repeatedly. Inspection of manufactured goods for quality control, surveillance of a traffic junction for road safety, and detection of abnormalities in an MRI scan are few of the examples where machine vision is extremely useful. While researchers have extensively been working to develop better and faster image processing algorithms to do more complex and demanding jobs, the rapid growth by the semi-conductor industry to make cheaper and faster computers has completely revolutionized machine vision. Nowadays, a lot of processes can be done fast because of the quick feedback provided by the state-of-the-art image processing algorithms running on state-of-the-art computers. Through this module we learn the basic processes involved in image processing and we apply some of this knowledge to process two different types of images.

Overview

This report comprises of two parts that describe our strategy to perform the required operations on the two different images. The image processing codes were first implemented in Octave (because of its lower memory requirement), and after all the algorithms worked as expected they were tested for compatibility with MATLAB. We tried to implement most of the functions on our own, and as a result they are not optimized and take longer to run. The codes are also available on www.github.com.

For both the images we are required to do the following operations -

1. Display the original image.
2. Create binary image using thresholding.
3. Separate and identify different characters.
4. Rotate the characters clockwise by 90 degrees.
5. Rotate the characters anti-clockwise 35 degrees.
6. Find the boundary of all the characters.
7. Find the one-pixel thin image of all the characters.
8. Rearrange the characters in a given sequence.

For most of the operations, we discuss a few different methods, to the best of our knowledge, that can be used to obtain the desired result with their respective advantages and disadvantages.

Processing of Image 1 - charact1.txt

Display the original image

The original image is in the form of a text file ‘charact1.txt’ (Figure 1.1). We use the inbuilt function ‘`fileread()`’ to read the contents of the input file as a vector of characters. This vector is then passed as an argument to a custom defined function ‘`textTOascii()`’. The `textTOascii` function creates a new empty vector, and then goes through each of the characters of the input vector one-by-one. Then the character is converted to its corresponding ASCII value by converting it to a double data type which converts the character to a numeric value in the range 0-255. If the ASCII character does not have a value of 10 (corresponds to new line) or 13 (corresponds to carriage return), it is valid and is appended to the empty vector. At the end of this operation we get a new vector of numbers of length 4096 elements.

It is important to note that this vector of characters can be converted into numbers by building a custom look-up table which links each character to a unique intensity value. For example, we could assign all the ‘0’ characters to have intensity value 0, all the ‘A’ characters to have intensity value 40, all the ‘R’ characters to have intensity value 100, and so on. But we chose to use the ASCII value as a look-up table because of the ease of converting character to its corresponding ASCII value and it maps all the characters in the image to a unique intensity value.

Next, the problem statement says that the image is 64 pixels x 64 pixels, and the vector of numbers is reshaped accordingly. The first 64 elements of the vector form the first row of the image, elements 65 to 128 form the second row of the image, elements 129 to 192 form the third row of the image and so on. This generate a grayscale image, with 32 unique intensity values with minimum and maximum intensity values of 48 (corresponds to character 0) and 86 (corresponds to character V) respectively (Figure 1.2). We observe that the background is darker than the foreground which comprises of the letters A, B, C and numbers 1, 2, 3.

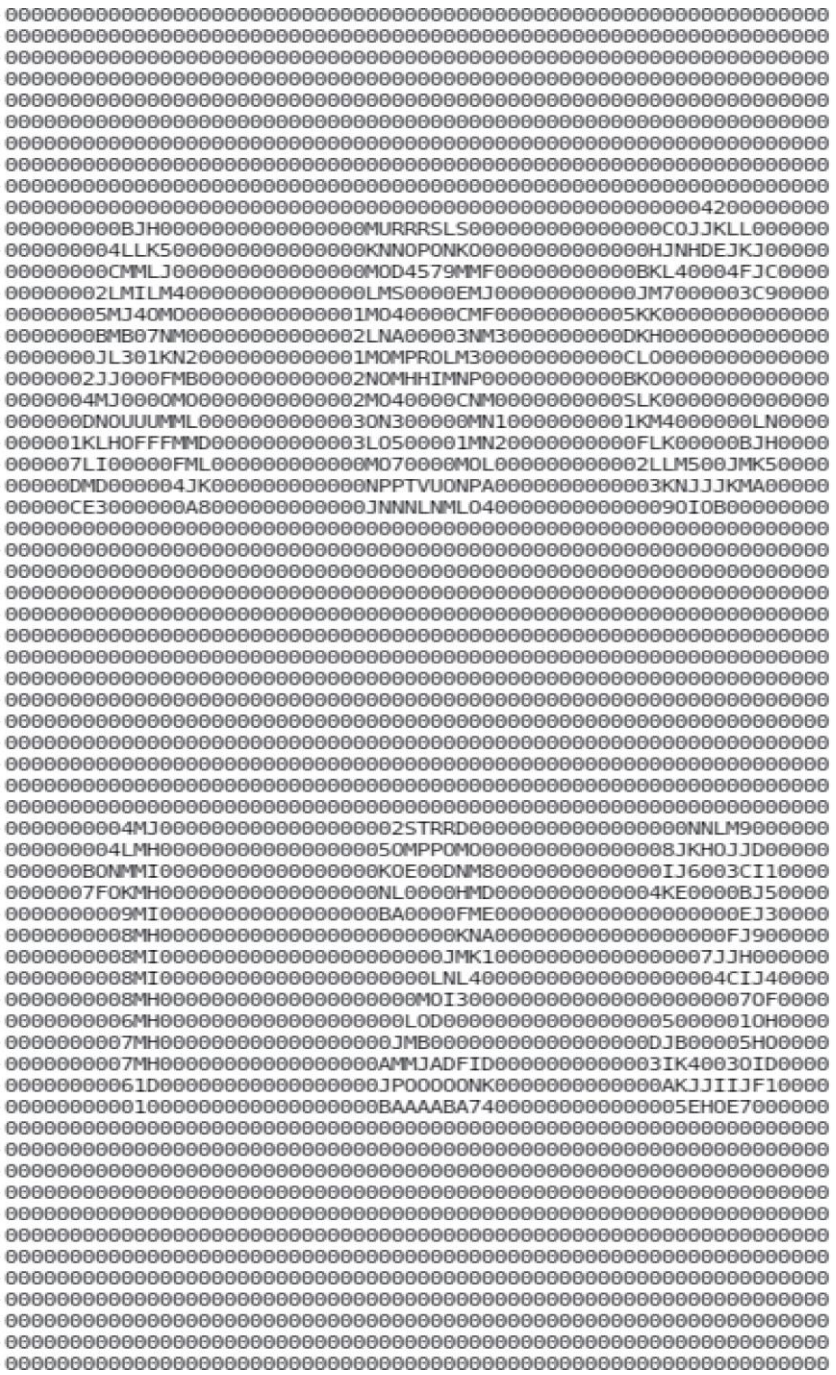

Figure 1.1 Input image in the form of text file. The text file consists of 32 different characters starting from 0 (ASCII code 48) to V (ASCII value 86). While parsing the character in the text file, return and new line characters (ASCII value 10 and 32) are ignored.

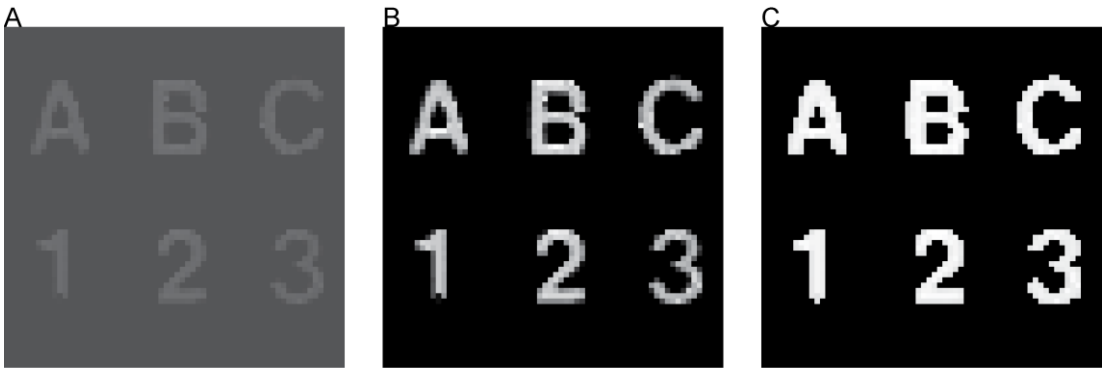

Figure 1.2 Contrast adjustment. (A) The text file when converted to an image using ASCII encoding. The minimum intensity value is 48 and the maximum intensity value is 86. **(B)** Contrast stretching of A using the rule that intensity value of 48 goes to 0 (black), intensity value of 86 goes to 255 (white) and the intermediate intensity values are linearly stretched between 0 and 255. **(C)** Image A after histogram equalization.

Create binary image using thresholding

If we observe the greyscale intensity values of the image, we see that all the background pixels correspond to intensity value of 48, and the foreground pixels value are higher. Thus, we can use this information to our advantage and threshold the image using different strategies which are discussed below.

1. **Constant threshold** - We can set up a global/constant threshold as 48. Then we scan through all the pixels of the image and if the pixel value is less than or equal to 48 it is set as 0 (background) but if it greater than 48 it is set as 1 (foreground/object). This process will be very quick as we need to scan through all the pixels just once and we will get a binary image after this operation. However, the problem with this step could be its generality. For example, if we decide to enhance the contrast of the image by histogram stretching or histogram equalization the same global threshold value might not work, and it will have to be changed. This is implemented in our '`threshold()`' which takes in the image, threshold method, and threshold value as parameters. The function returns a thresholded binary image and the set threshold value. To use the function in constant threshold mode, type `threshold(img, 'constant', 48)`.

2. **Mean threshold** - We can calculate the average intensity value from all the pixels and set it as the threshold. For the image in Figure 1.2A generated using the ASCII character mapping, the average intensity value is 50.660. All the pixels less than or equal to 50.660 are set as 0 and those greater than 50.660 are set as 1. This method is slower because we need to scan through all the pixels twice. In the first run, we calculate the mean intensity value and in the second run we apply it as a threshold. Even though the mean threshold method is slower it is more robust to some point processing operations like histogram stretching and equalization. If the background pixels remain darker then the foreground pixels, the mean threshold will work reasonably well for similar images. To use the function in mean threshold mode, type `threshold(img, 'mean')`.

3. **Median threshold** - Just like the mean, median is also a number which describes the central tendency of the intensity values of the image. The

median intensity value for the image in Figure 1.2A is 48 and the binary image is generated accordingly. To use the function in mean threshold mode, type `threshold(img, 'median').`

4. **Otsu and maximum entropy (Kapur) threshold** - These are more modern thresholding techniques which work best on a bimodal distribution. The complete details of these methods are out of the scope of this report, but they work on the principle of minimizing the inter-class-variance (Otsu's method) or maximizing the inter-class entropy (Kapur's method) between the two distributions which make up the foreground and the background. For the image in Figure 1.2A, the calculated Otsu and Kapur threshold values are 57 and 48 respectively. To use the function in Otsu threshold mode, type `threshold(img, 'otsu')`. To use the function in Kapur threshold mode, type `threshold(img, 'maxentropy')`.

It should be noted that higher the threshold value, smaller are the foreground objects. For example, a threshold value of 48 will give the larger objects compared to if we use threshold value of 50.660. In Figure 1.3 we present the binary image generated using different threshold methods.

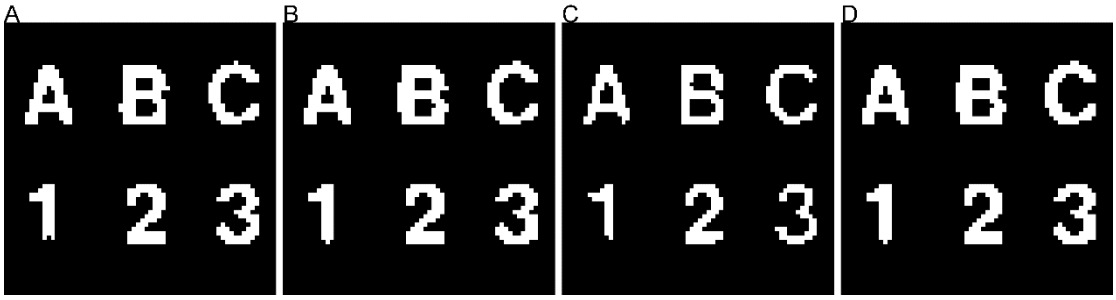

Figure 1.3 Comparison of different thresholding methods. (A) Mean intensity value is calculated as 50.660 and is used as threshold. **(B)** Median intensity value is 48 and it is used as threshold. **(C)** Otsu's threshold is evaluated to be 57 and is the largest of the 4 different methods used. **(D)** Kapur's threshold is evaluated as 48 and the binary images generated using this threshold is same as B. We can observe that as the threshold value increases the detected objects become smaller. Otsu's threshold is the largest and the objects in C are smallest.

Separate and identify different characters

Now, we have a binary image where the pixels containing the letters and alphabets are marked as 1 and the background is marked as 0. Then we do a connected component analysis where we scan through the image and group the pixels into components based on the pixel connectivity. For example, in Figure 1.3A the all the pixels representing alphabet 'B' have intensity 1. We start from one such pixel and start labeling all the 1 value pixels which are directly 8-connected to that pixel and give it the same label. This analysis is repeated for the whole image until all the pixels are assigned a label. After this operation, the generated labeled image has 7 unique labels. 0 corresponds to the background, and labels 1, 2, 3, 4, 5, 6 correspond to 'A', '1', 'B', '2', 'C', '3' respectively. We use the MATLAB inbuilt function '`bwlabel()`' to perform this labeling using 8-connected neighbors.

We do a histogram stretching on the labeled image to change the range of intensity values from 0-7 to 0-255 to make it visually appealing and present it in Figure 4A. We also use another image processing software Fiji to represent the labels using different colors (Figure 4B).

From the labeled image, we can extract more features like the center, bounding box, area etc. for each of the labels. A bounding box is the smallest rectangle, oriented along the rows and columns of the image, which can fully contain the desired region of interest (ROI). In our case, the ROI is each label of the foreground object that we get from the labeled image. Once we get the bounding box, we treat its center as the center of the object, the width of the box corresponds to the width of the object and the height of the box corresponds to the height of the object. The total number of pixels that have the same label is defined as the area of that object. For example, in the labeled image (Figure 1.4) alphabet 'A' has a label 1 and there are 84 pixels with the same label. So, the area of alphabet 'A' is 84. Similarly, number '2' has label 4 and number of pixels with label 4 are 77. Hence, the area of number '2' is 77.

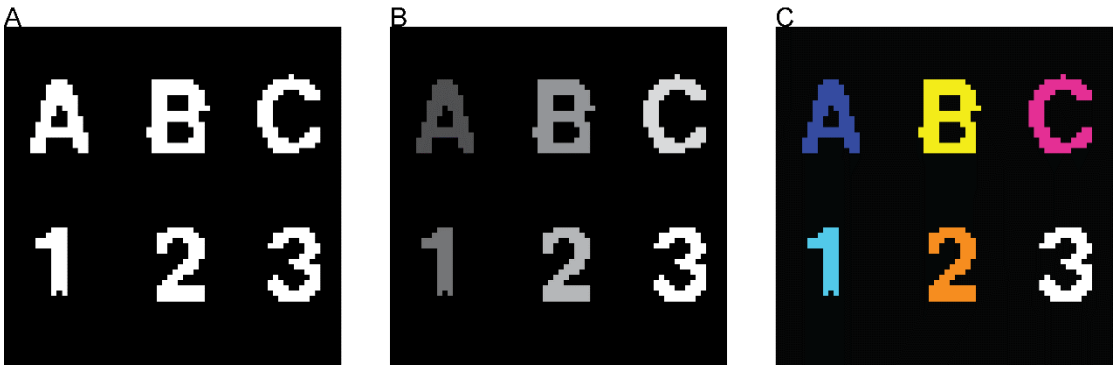

Figure 1.4 Labeling the connected components. (A) Binary image generated using mean threshold. **(B)** There are 6 objects in the image, and they are given labels from 1 to 6. To display the images clearly, we do a contrast stretching to utilize the whole grayscale. Background is set as 0, all the pixels with label 6 are set to 255 and the intermediate values are linearly spread in between 0 to 255. From the grayscale level, we can tell than 'A' is object 1, '1' is object 2, 'B' is object 3, '2' is object 4, 'C' is object 5, and '3' is object 6. **(C)** To make the visualization better, we use a different colormap to represent the labeled image.

Using the information from bounding box, we can crop smaller sections from the original grayscale image and plot the different characters separately (Figure 1.5). Here we arrange all the characters in the sequence of their labels mentioned earlier.

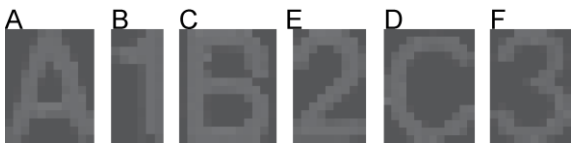

Figure 1.5 Labels for different connected components. We draw bounding boxes around each of the labeled objects, and using the coordinates of the bounding box, we extract the respective regions from the original grayscale image. **(A-E)** All the ROIs are cropped and saved individually according to their label and put together using Adobe Illustrator software. 'A' is object 1, '1' is object 2, 'B' is object 3, '2' is object 4, 'C' is object 5, and '3' is object 6.

To find the properties of labeled region, we can use the following two functions.

```
[labelImg, numLabel] = bwlabel(bImg, 8);

props = regionProps(labelImg);
```

The first function ‘bwlabel()’ takes in the binary image and pixel connectivity as input. Here we use 8-connected neighbors to label different objects in binary image ‘bImg’. The function returns the labeled matrix ‘labelImg’ and the number of objects ‘numLabel’ that were found in the binary image. Then we use the custom written ‘regionProps()’ function which takes in the labeled matrix as input and finds out the bounding box, area, center, width, and height of each of the connected components.

Rotate the characters clockwise by 90 degrees

Rotation of each of the characters clockwise can be done in two ways.

1. We can think of rotation a matrix by 90 degrees clockwise as doing a transformation which takes the first row of a matrix to the last column, second row to the second last column, third row to the third last column and so on. If the input matrix has dimension $M \times N$ then the final transformed matrix has dimension $N \times M$. Even though this can be very fast and easy to implement, we did not use it because it can be used only for this specific purpose.

2. In general, a rotation matrix is defined as

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Here, θ is the angle of rotation in the counter clockwise direction. So, if we want to rotate an object by 90 degrees clockwise, it would be equivalent to rotating the object by -90 degrees counter clockwise. Hence, we can use the above transformation by setting θ = -90 degrees. We choose to use this approach because it can be used very easily for the next part of the problem where we are required to rotate the image by 35 degrees counter clockwise.

We write our custom function ‘imageRotate()’ which takes in three parameters. First one is the input image, second is the angle in degrees by which we want to rotate the image, and the third one is the interpolation method to be used after rotation. We implemented the nearest neighbor and the bi-linear interpolation methods. Nevertheless, for rotation by 90 degrees using any interpolation method produces same result as all the pixel position, after transforming through the rotation matrix get mapped to integer pixel values and hence bi-linear interpolation becomes redundant. For this specific problem, we use `imageRotate(cropImg, -90, ‘nearestNeighbor’)`. Figure 1.6 shows the image where each character is rotated around its center by -90 degrees.

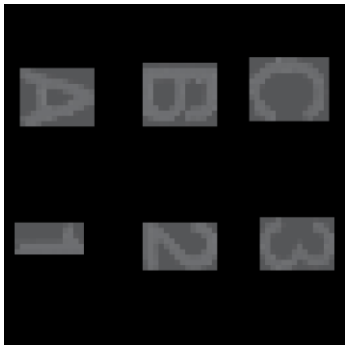

Figure 1.6 Image rotation 1. Each of the grayscale characters in Figure 1.5 is rotated clockwise by 90 degrees and placed on a blank 64x64 image. The ROI is placed on the blank image in such a way that the rotation of object seems in place. It should be noted that we use the nearest neighbor interpolation method here. But using bilinear interpolation method will give the same result as all the pixels, after rotation get mapped to an exact integer coordinate and in this case nearest neighbor and bilinear interpolation methods are equivalent.

Rotate the characters counter-clockwise 35 degrees

We use the same `'imageRotate()'` function with input rotation angle as 35 degrees and generate two images for two different interpolation methods. Figure 1.7 compares the nearest neighbor interpolation with the bi-linear interpolation. It should be noted that the bi-linear interpolation is faster than the bi-linear interpolation method. We also tried implementing bi-cubic interpolation method discussed in the lectures, but our implementation was not correct.

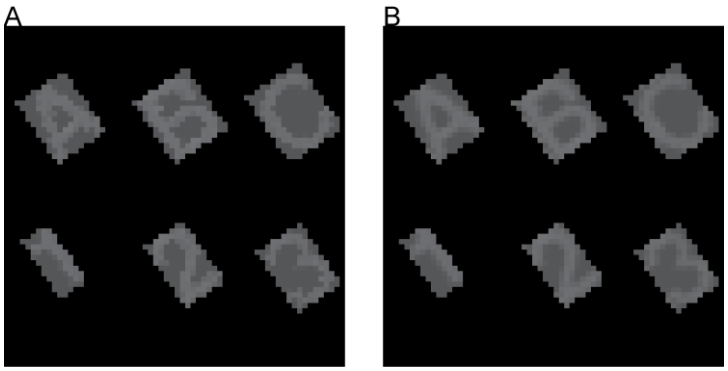

Figure 1.7 Image rotation 2. Each of the grayscale characters in Figure 1.5 is rotated counter clockwise by 35 degrees and placed on a blank 64x64 image. The ROI is placed on the blank image in such a way that the rotation of object seems in place. **(A)** Nearest neighbor interpolation method is faster but gives a rough profile after rotation. **(B)** Bilinear interpolation method is slightly slower, but it can be observed that the characters seem smoother (Compare characters 'A' and 'B' in both the images).

Find the boundary of all the characters

The boundary for a binary can be found in two steps. First, we erode the binary image using an all 1's square structuring element of size 3. Next, we subtract the eroded image from the original binary image. This gives up the boundary pixels. Even though other methods to find the object boundary can be found in literature, we used this because it is easy to implement. The boundary of all

the characters is shown in Figure 1.8. Since the characters in this image are very thin, the boundary looks fat in some places (middle section of character ‘B’). Since this section was just 2 pixels wide in the original binary image, it got eroded from the top and bottom. Thus, this region was fully eroded and has all 0-pixel values after erosion. Following this when we subtract the eroded image from original image, the two-pixel thick feature come back in the same place.

We implement this operation in `‘boundary()’` and it can used as `‘bImgBdry = boundary(bImg)’`. `‘bImg’` is the binary image for which we need to find the boundary and the function return a new binary image which contains the boundary for all the components of the input image.

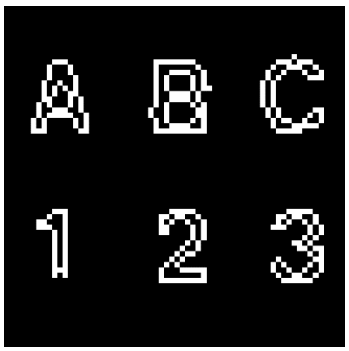

Figure 1.8 Boundary of binary image. To find out the boundary of binary image we perform a morphological erosion operation using a square structuring element (all ones) of size 3. This erosion operation removes the pixels at the edge of each of the binary components. Finally, the eroded image is subtracted from the original binary image and only the pixels which were at the boundary of all the objects remain 1. All the interion points become 0 and the background also stays 0.

Find the one-pixel thin image of all the characters

A binary object can be thinned down using by eroding with certain morphological operators in a sequential way. Figure 1.9 highlights the structuring elements we used for thinning. If we look carefully at the the top row of kernels, they are similar except that in each step they have rotated by 90 degrees clockwise. Same observation holds true for the bottom row of kernels. To thin down a binary object, we erode it using these kernels in the following sequence A1 → B1 → A2 → B2 → A3 → B3 → A4 → B4. After the first erosion, we compare the eroded image with the original. If the images are different, we do the same erosion operation on the eroded image again in the same sequence. Finally, we will get an image which does not change by performing erosion using these operators. Then we call it the thinned image.

We implement the thinning in our custom written function `‘skeleton()’`. This function takes in a binary image as input and keeps eroding it using kernels in Figure 1.9 until it gets a stable skeleton and then it returns the thinned image. To use this function `‘bImgSkeleton = skeleton(bImg)’`. Here `‘bImg’` is the input binary image and `‘bImgSkeleton’` is the result of skeletonization. The result of this is shown in Figure 1.10.

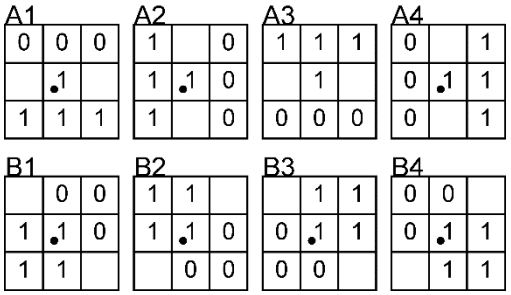

Figure 1.9 Structuring elements for thinning. To skeletonize a binary image, we use the above structuring elements in this sequence (A1 → B1 → A2 → B2 → A3 → B3 → A4 → B4). After the image is eroded using once using all the kernels in the sequence above, the new image is compared with the previous version. If the images are same, then the skeletonization is done. If not, the new image is eroded through the same cycle of structuring elements. This process is repeated until we get the skeleton which does not change by the above erosion kernels. The black dot represents the origin of each structuring element.

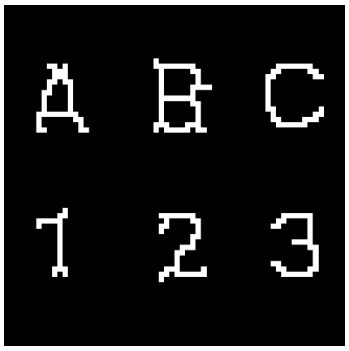

Figure 1.10 Skeleton of binary image. Result of thinning/skeletonization using the structuring elements described in Figure 1.9. Each region is 1 pixel thick and representative of the central skeleton of the binary objects.

Rearrange the characters in a given sequence

We have extracted the characters in previous step after performing segmentation and labeling. We find that all the characters are marked different objects. ‘A’ is object 1, ‘1’ is object 2, ‘B’ is object 3, ‘2’ is object 4, ‘C’ is object 5, and ‘3’ is object 6. In addition to that we also know the size of all the characters are –

A has 14 rows and 11 columns of pixels

B has 14 rows and 12 columns of pixels

C has 15 rows and 12 columns of pixels

1 has 13 rows and 6 columns of pixels

2 has 14 rows and 9 columns of pixels

3 has 14 rows and 10 columns of pixels

We find that character ‘C’ has the largest number of rows. And since we want to stack the images along columns, the number of rows of each image should be same. We rescale all the cropped regions using the ‘`imageScale()`’

function and set the target rows of the scaled image as 15. The number of target columns are chosen in such a way that the aspect ratio of the character is maintained. Then we concatenate the images together in the desired sequence.

Figure 1.11 Collate the characters together. The labeling function returns the labels of the characters as follows. ‘A’ is object 1, ‘1’ is object 2, ‘B’ is object 3, ‘2’ is object 4, ‘C’ is object 5, and ‘3’ is object 6. They need to be arranged in this label sequence 2, 1, 4, 3, 6, 5. However, in order to concatenate images next to each other the number of rows must be same. We find that character ‘C’ is the largest with 15 rows and we scale all the other cropped ROI such that all of them have 15 rows and the number of columns is calculated by imposing the condition that the aspect ratio of the ROI does not change.

Processing of image 2 – charact2.bmp

Display the original image

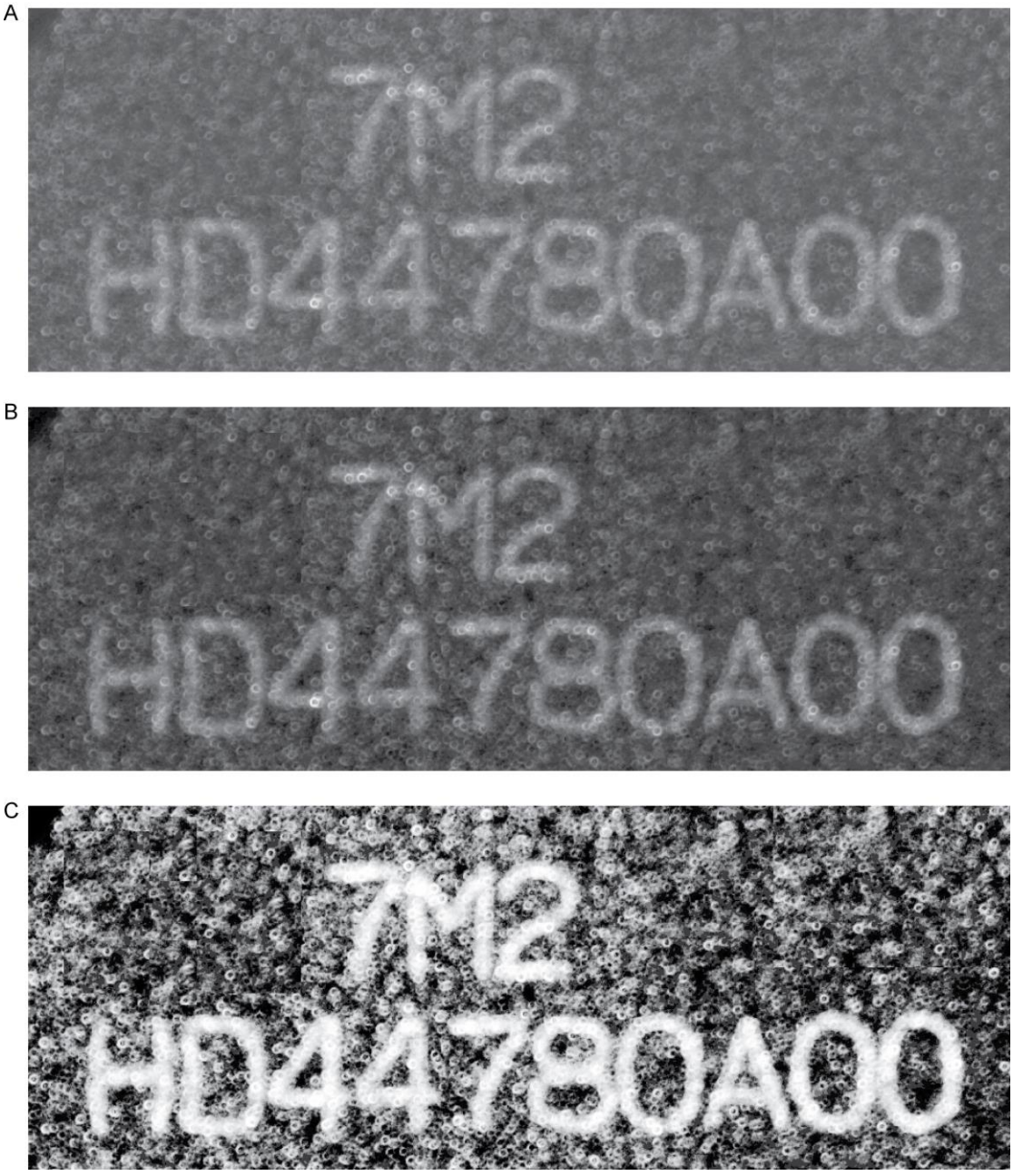

Figure 2.1 Read raw image. (A) The grayscale bitmap file with mimimum intensity value of 41 and maximum intensity value of 246. **(B)** Contrast stretching of A using the rule that intensity value of 41 goes to 0 (black), intensity value of 246 goes to 255 (white) and the intermediate intensity values are linearly stretched between 0 and 255. **(C)** Image A after histogram equalization.

We use inbuilt `'imread()'` to read the image in the RGB mode. But since it is a grayscale image the red, green, and blue values of each pixel is same. So, we keep only the red channel for processing.

Create binary image using thresholding

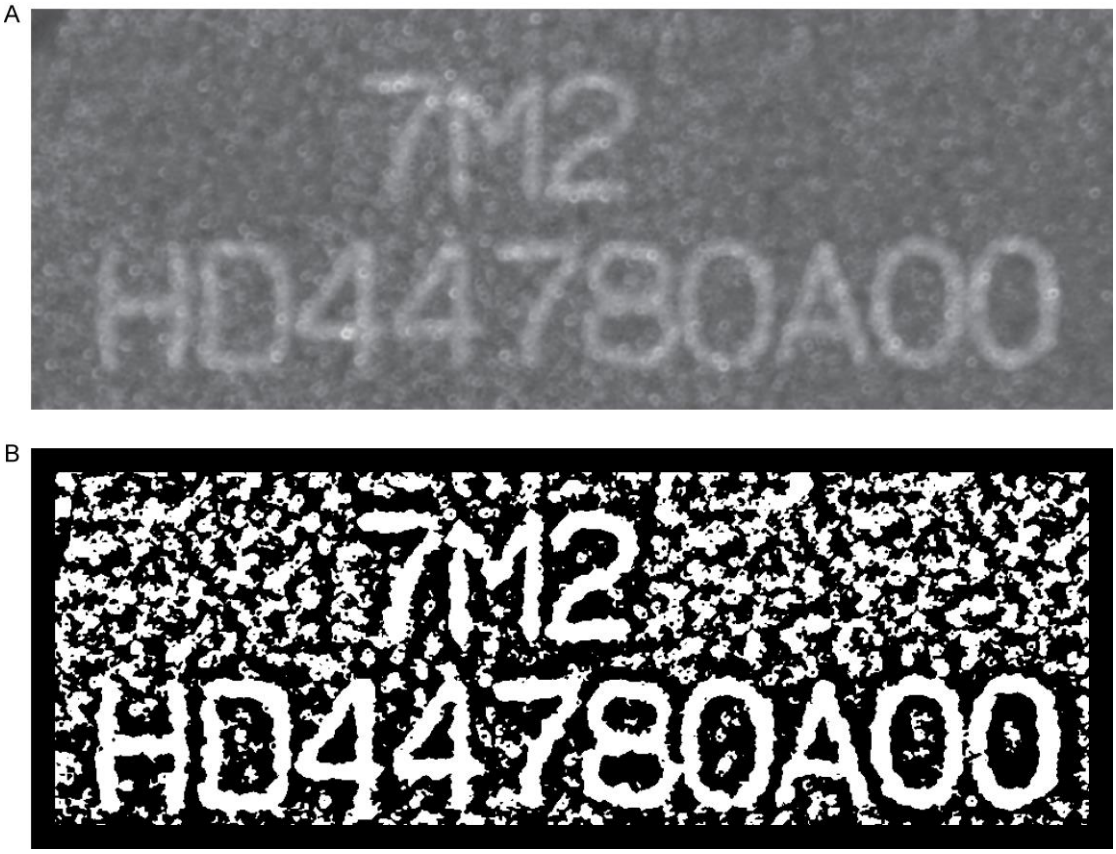

Figure 2.2 Adaptive thresholding for the image. (A) The image after the Gaussian blurring, using $\sigma = 1$ pixel. **(B)** The binary image obtained after the adaptive mean thresholding method with a kernel size of 45 pixels. The structuring element generated is an all 1s square with size 45.

The segmentation of this image is relatively difficult because of high noise in the background. We use the following sequence of steps for segmentation.

1. Blur the image slightly using the Gaussian filter with $\sigma = 1$ to remove the high frequency noise feature (Figure 2.2A).
2. Apply adaptive mean threshold to segment the characters along with background noise (Figure 2.2B)
3. Blur the image strongly using a Gaussian filter with $\sigma = 6$ pixels to make the image very smooth (Figure 2.3A).
4. Apply a global Otsu's threshold to find an optimal threshold value separating the bimodal distribution (Figure 2.3C). The optimal threshold value is calculated as 97. The binary image obtained after this thresholding does not have background noise, but the characters have merged to form a single object (Figure 2.3B).
5. Take a logical and of the two binary images (Figure 2.2B and Figure 2.3B). This generated a relatively clean binary image with most of the background filtered out (Figure 2.4A).
6. Do binary opening and closing operations with all 1s structuring elements of size 5 and 3 respectively to remove the sparse noisy segmented regions (Figure 2.4B).

7. Label the connected components of this binary image and do the next set of operations required in the project.

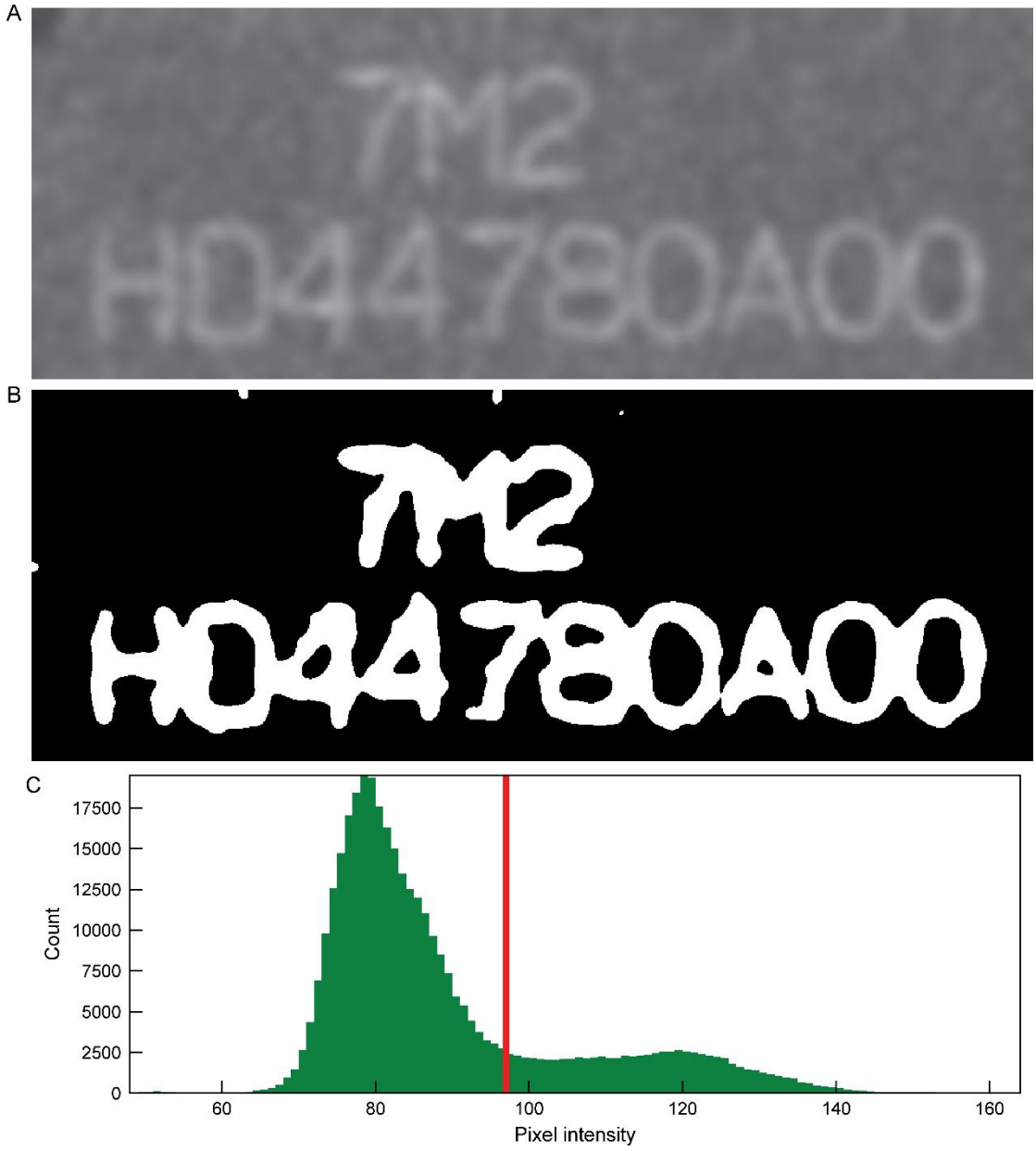

Figure 2.3 Global Otsu's thresholding. (A) The image after the Gaussian blurring with $\sigma = 6$ pixels. **(B)** The binary image obtained after the Otsu thresholding method. We observe that the background noise is small, but the characters touch each other. **(C)** The histogram of the image in A with obtained threshold value. Red line represents the Otsu's threshold value of 97. Note that most of the intensity values lie in the range 60 to 140.

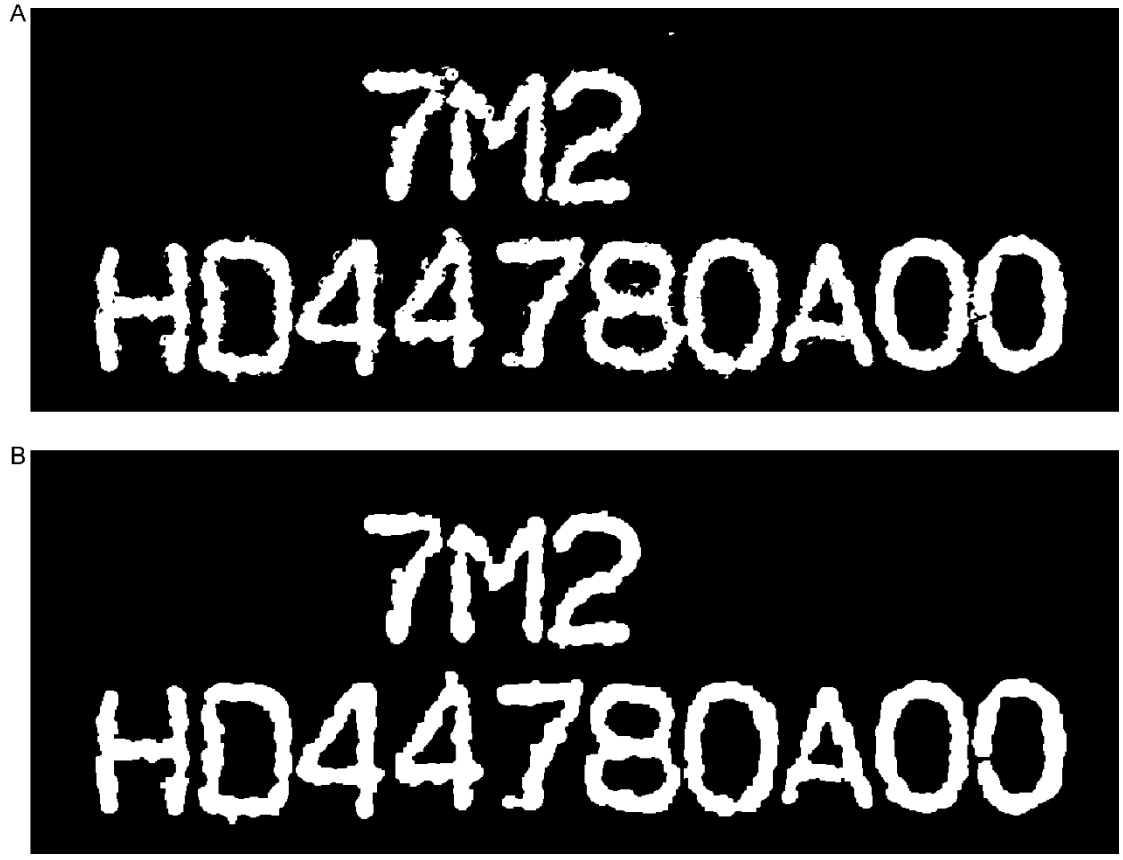

Figure 2.4 Refining the binary image. (A) The logical AND operator was applied on the binary images after applying the Otsu's (Figure 2.3B) and Adaptive thresholding method (Figure 2.2B). We see some small spots and rough edges around the detected object. These can be cleaned using binary morphological operations. **(B)** Opening and closing morphological operators were applied with all 1s square kernels with size 5 and 3 respectively.

Separate and identify different characters

Figure 2.5 Labeling the connected components. (A) A labelled image with corresponding gray-level values assigned to each region, using 8-connetivity. There are 13 objects in the image, and they are given labels from 1 to 13. **(B)** False-color representation of the labelled image to make it visually more appealing.

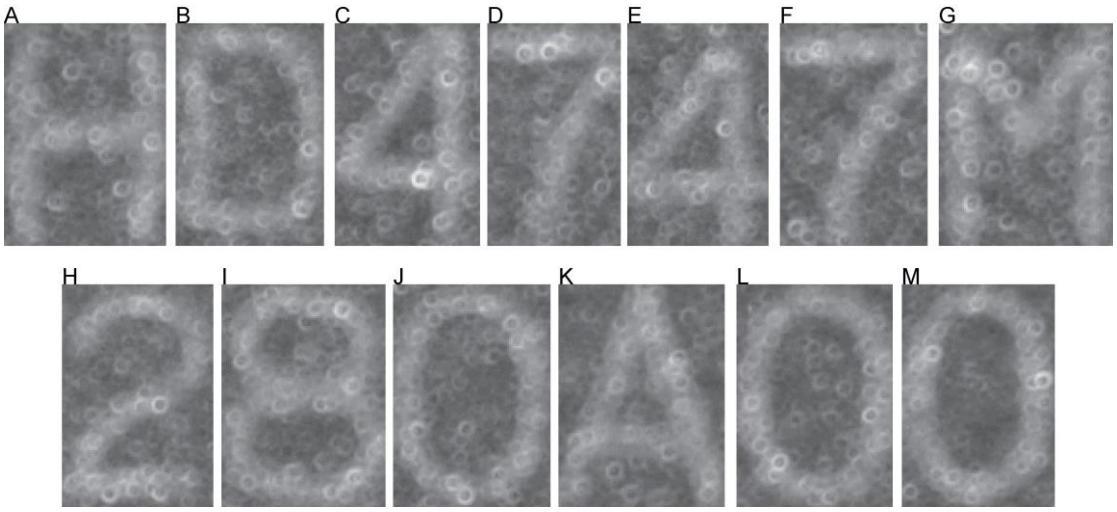

Figure 2.6 Isolating each character in the image. We draw bounding boxes around each of the labeled objects, and using the coordinates of the bounding box, we extract the respective regions from the original grayscale image. **(A-M)** All the ROIs are cropped and saved individually according to their label and put together using Adobe Illustrator software. ‘A’ is object 1, ‘D’ is object 2, ‘4’ is object 3, ‘7’ is object 4, ‘4’ is object 5, and ‘7’ is object 6, ‘M’ is object 7, ‘2’ is object 8, ‘8’ is object 9, ‘0’ is object 10, ‘A’ is object 11, ‘0’ is object 12, ‘0’ is object 13.

Rotate the characters clockwise by 90 degrees

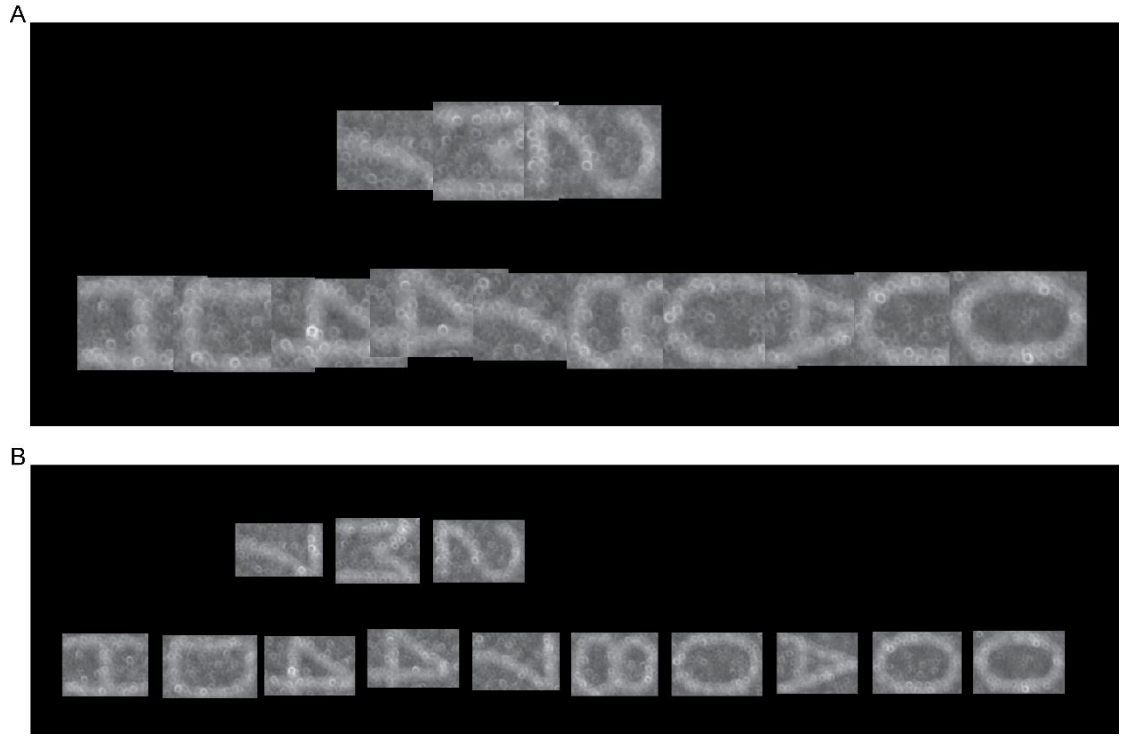

Figure 2.7 Image rotation 1. (A) The rotated objects in the image around its centroid clockwise direction by -90 degrees. Since the objects are very close to each other, they overlap after rotating. **(B)** To remove the overlap, we create a new blank image which is larger than the original image along columns. Then we place the rotate object in such a way that they do not overlap. For example, if we want to place ‘M’ at its original centroid position, part of the rotated image will fall on ‘7’. So, we translate ‘M’ by 50 pixels along the x axis until it stops overlapping with ‘7’. In the same way each object is rotated and separated from each other. It should be noted that we use the nearest neighbor interpolation method here. But using bilinear interpolation method will give the same result as all the pixels, after rotation get mapped to an exact integer coordinate.

Rotate the characters counter-clockwise 35 degrees

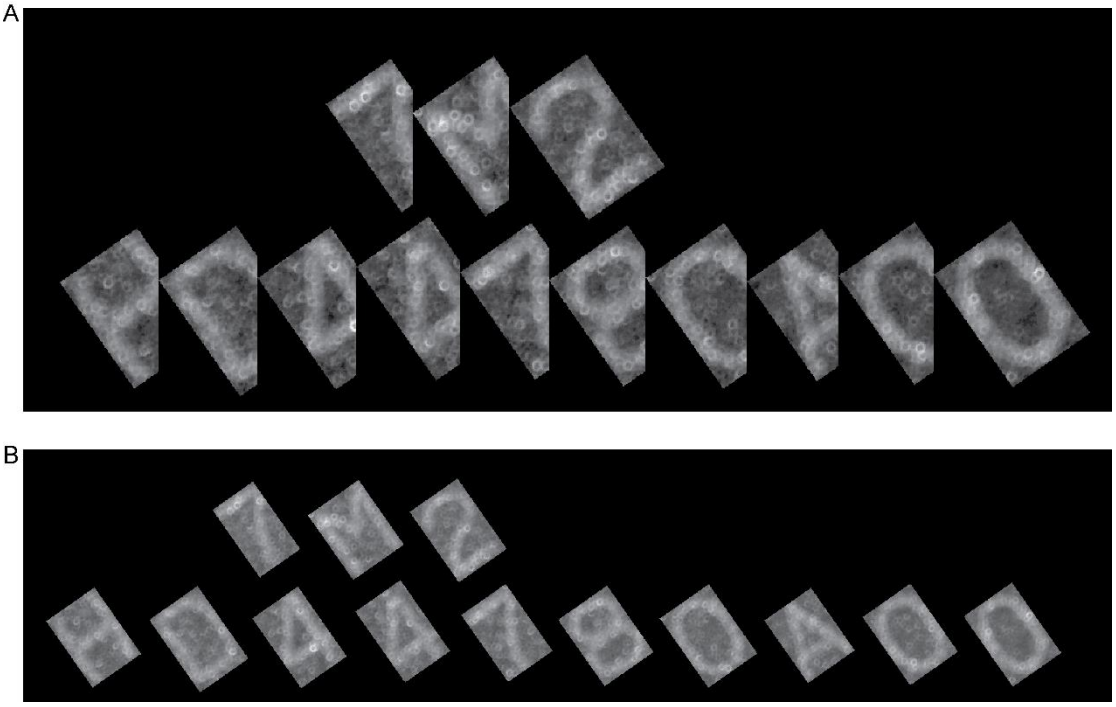

Figure 2.8 Image rotation 2. (A) The rotated objects in the image around its centroid clockwise direction by 35 degrees. Since the objects are very close to each other, they overlap after rotating. **(B)** To remove the overlap, we create a new blank image which is larger than the original image along columns. Then we place the rotate object in such a way that they do not overlap. For example, if we want to place 'M' at its original centroid position, part of the rotated image will fall on '7'. So, we translate 'M' by 20 pixels along the x axis until it stops overlapping with '7'. In the same way each object is rotated and separated from each other. It should be noted that we use the nearest neighbor interpolation method here. We can also use the bilinear interpolation method, but we show results based on the nearest neighbor interpolation only.

Find the boundary of all the characters

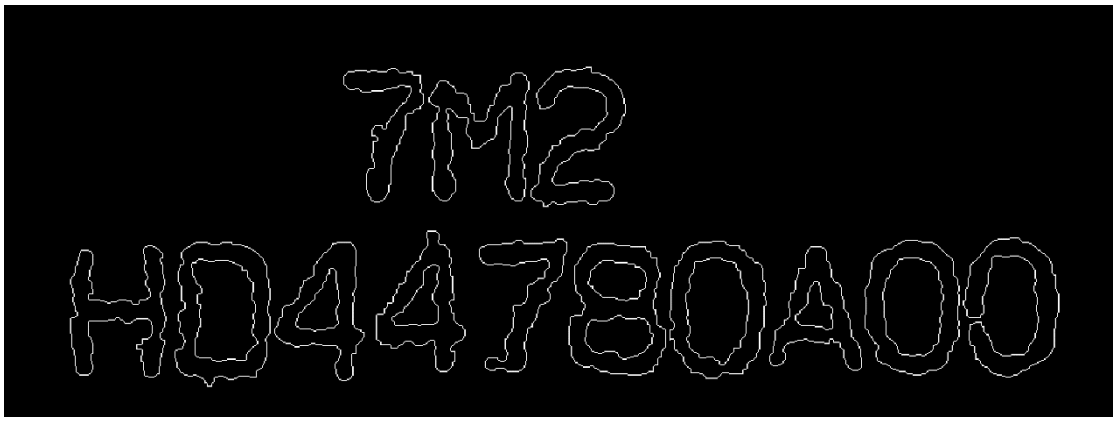

Figure 2.9 Boundary of binary image. To find out the boundary of binary image we perform a morphological erosion operation using a square structuring element (all ones) of size 3. This erosion operation removes the pixels at the edge of each of the binary components. Finally, the eroded image is subtracted from the original binary

image and only the pixels which were at the boundary of all the objects remain 1. All the interior points become 0 and the background also stays 0.

Find the one-pixel thin image of all the characters

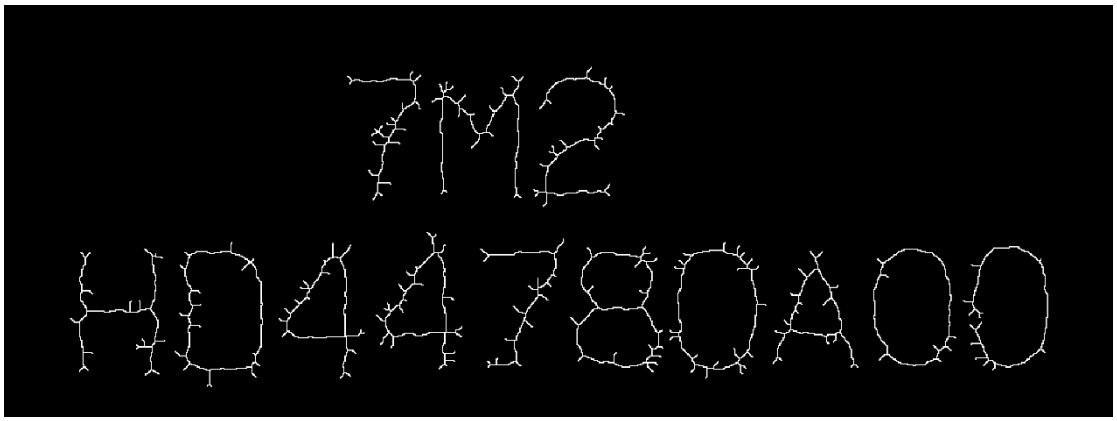

Figure 2.10 Skeleton of binary image. Result of thinning/skeletonization using the structuring elements described in Figure 1.9. Each region is 1 pixel thick and representative of the central skeleton of the binary objects.

Rearrange the characters in a given sequence

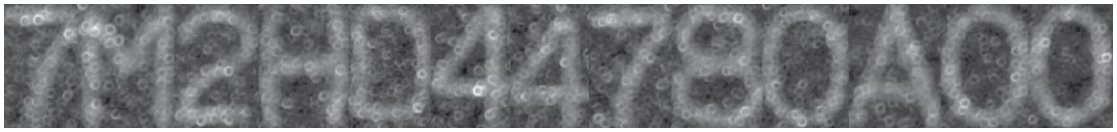

Figure 2.11 Collate the characters together. The labeling function returns the labels of the characters as follows. ‘A’ is object 1, ‘D’ is object 2, ‘4’ is object 3, ‘7’ is object 4, ‘4’ is object 5, and ‘7’ is object 6, ‘M’ is object 7, ‘2’ is object 8, ‘8’ is object 9, ‘0’ is object 10, ‘A’ is object 11, ‘0’ is object 12, ‘0’ is object 13. They need to be arranged in this label sequence 4, 6, 8, 1, 2, 3, 5, 7, 9, 10, 11, 12, 13. However, in order to concatenate images next to each other the number of rows must be same. we scale all the other cropped ROI such that all of them have the same rows and the number of columns is calculated by imposing the condition that the aspect ratio of the ROI does not change.

Conclusion

We were able to successfully implement most of the functions on our own. However, our implementation does not take advantage of the power matrix algebra that can be performed very efficiently in MATLAB. This is definitely one area of improvement. In addition to this, most of the development was done using Octave and we noticed that the implementation of the Gaussian blurring is different in MATLAB and Octave. As a result, the segmentation result for image 2 in MATLAB is slightly different from what we show in the report. However, the same algorithm works well for both the software.

We tried to keep our codes updated in and track changes using the git version control system and we plan to update the repository in the future with implementation of more complex algorithms.

Keeping this in mind, these two images could have processed in a lot of different ways. For example, we could have chosen to use median threshold instead of the mean for image 1, but the nature of results would have been very similar. In addition to this, the speckled noise in segmentation of image 2 could have been removed using an area filter which retains the connected components only in a certain area range and remove all the extra white pixels. These are a few other methods which could certainly be explored.

References

1. Gonzalez, Rafael C., and Richard E. Woods. "Digital image processing." (2002).
2. Kovesi, Peter D. "MATLAB and Octave functions for computer vision and image processing."
3. Solomon, C., & Breckon, T. (2011). Fundamentals of Digital Image Processing: A practical approach with examples in Matlab. John Wiley & Sons.
4. Abràmoff, M. D., Magalhães, P. J., & Ram, S. J. (2004). Image processing with ImageJ. Biophotonics international, 11(7), 36-42.
5. Schneider, C. A., Rasband, W. S., & Eliceiri, K. W. (2012). NIH Image to ImageJ: 25 years of image analysis. Nature methods, 9(7), 671.
6. https://github.com/uanand/ME5405.git
7. Pal, N. R., & Pal, S. K. (1993). A review on image segmentation techniques. Pattern recognition, 26(9), 1277-1294.
8. Vala, Hetal J., and Astha Baxi. "A review on Otsu image segmentation algorithm." International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) 2.2 (2013): 387-389.
9. Wang, Y., Chen, Q., & Zhang, B. (1999). Image enhancement based on equal area dualistic sub-image histogram equalization method. IEEE Transactions on Consumer Electronics, 45(1), 68-75.
10. https://homepages.inf.ed.ac.uk/rbf/HIPR2/hipr_top.htm
11. Otsu, N. IEEE Transactions on Systems, Man and Cybernetics 1979,9, 62-66.
12. Rasband, W. https://imagej.nih.gov/ij/1997,1997-2016.