ME5405 Computer Project Report

CUI Hongjian A0133997M

LIU Congyi A0147968H

WANG Song A0143059H

1. [**Introduction**](#_ws8v2xyjoi68) **3**
2. [**Description of Algorithm**](#_e6712r31hfka) **5**

[Algorithm for Image Display & Binarization](#_d2gwcobu8s00) 5

[Algorithm for Locating Outline](#_ahpterlz3fgw) 5

[Algorithm for Segmentation](#_9r2pns96mr88) 6

[Algorithm for Segment Rotation](#_mdrrpabwpc00) 10

[Algorithm for Thinning Operation](#_vmbl5ik8u8q2) 10

1. [**Results**](#_ukgi8yi0ppz) **12**

[Image 1](#_4zr14oh2bnlx) 12

[Image 2](#_ur3clsqnsmt6) 13

1. [**Assessment of Methods**](#_108un9htpd1u) **15**

[Thresholding](#_h4tmntx0md08) 15

[Segmentation](#_lybwiv7thy0h) 16

[Thinning](#_jeehv2jvpf1p) 16

1. [**Conclusion**](#_oos1oiijxv2k) **18**
2. **References 18**

# **Introduction**

Digital image processing refers to manipulations of digital images using computer algorithms. As a subdivision of digital signal processing, digital image processing has wide applications in areas such as traffic control, navigation, astrophysics, biomedical imaging and so much more. The wide opportunities of digital image processing is only available to us learners if we have a solid understanding of the fundamentals. In this report, key concepts including thresholding, morphological transformation and segmentation are learnt, as well as implemented using MATLAB programme. A series of key image processing tasks are listed in Table 1.

| Task No | Task Statement | Key Points |
| --- | --- | --- |
| 1 | Display the image on the screen | - |
| 2 | Threshold the image and convert it into a binary image | Global thresholding |
| 3 | Determine the outlines | Outline boundary pixels |
| 4 | Segment the image, and identify the different characters | Connected component algorithms |
| 5 | Rotation of the image about their perspective centroid by 90 degree clockwise | Rotation of each segment |
| 6 | Rotation of the image about their perspective centroid by 30 degree counterclockwise | - |
| 7 | Determine a one-pixl thin image of the characters from step 4 | Thinning, morphological transform |
| 8 | Scale and display the characters in Image 1 in the order of A1B2C3 | - |
| 9 | Scale and display the characters in Image 2 in one line in the sequence **7M2HD44780A00** | - |

Table 1. Key tasks for this project

Two simple images, ‘charact1.txt’ and ‘charac2.bmp’ is used as the stock images for the processing, which are shown in Fig1. and Fig 2, detailed explanations of the algorithm are discussed in Section II. Screenshots of the result of each steps are listed in Section III, and further explanations and assessment of methods used here are elaborated in Section IV. In the last Section, an conclusion plus an comparison of image processing procedures for Image 1 and Image 2 is presented.

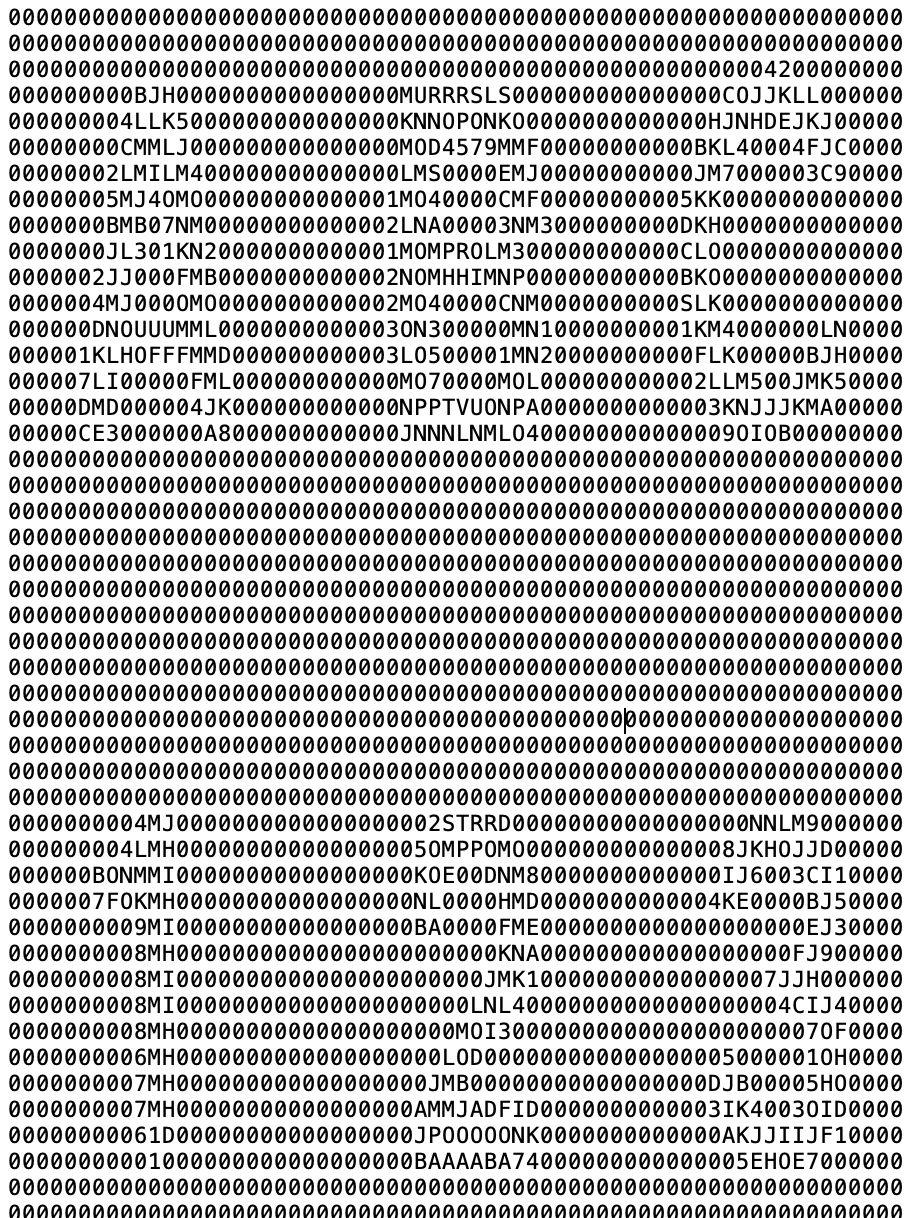


Fig 1. A screenshot of ‘charact1.txt’

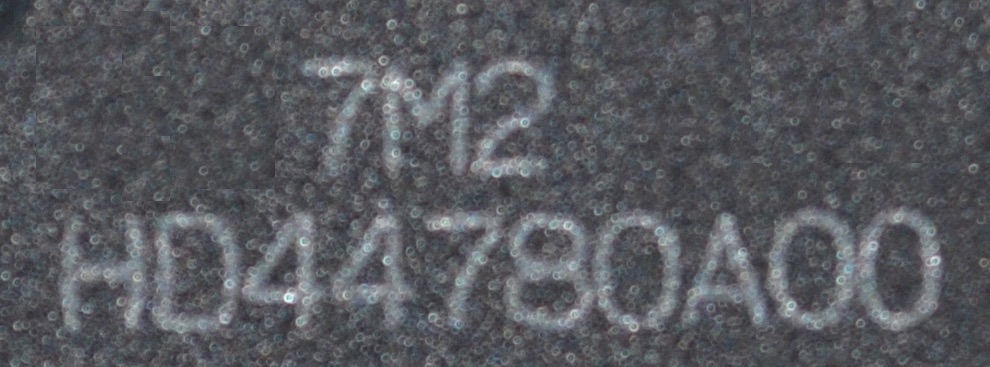


Fig 2. ‘charact2.bmp’

# **Description of Algorithm**

As an overview, there are two separate codes in the zip file that readers can refer to. They are named ‘test01.m’ and ‘test02.m’, which deals with tasks in image 1 and image 2 separately in the order of the task statements. Most functions that are discussed in the following are the same in both files, with minor difference mainly in task 1, 2 and task 8, 9. You can refer to either file alongside with the explanations in the following, unless there are special reference.

## Algorithm for Image Display & Binarization

Image 1 is txt file that contains characters instead of a matrix of pixel values. Function fopen is called to read the txt file, followed by another inbuilt function fscanf to convert the data into an array. As fopen reads data in a column order, a transpose is needed to put the image in the correct orientation. A further scan of the resultant matrix is performed, and all non-zero character is replaced by 1. Hence a matrix consisted of only 1 and 0 is obtained, and that is a binary image.

For Image 2, bmp can be readily read by imread function that is native in MATLAB. A colored image is N\*N\*3 matrix, which each pixel is consisted of three RGB values. To convert this picture to binary, the first step is to convert the colored image to a grayscale image by function rgb2gray, and the image is reduced to a matrix N\*N.

Before the binarization step, an opening operation is applied to the grayscale image. The reason for that is the excessive noise in image 2 interferes with the subsequent segmentations. An opening operations is to erode the image with a particular structuring element, followed by a dilation of the same structuring element. Noise pixels that is sufficiently small would be eroded away, and therefore disappear in the dilation step, are effectively filtered away. After some trials with different structuring element, a square strel of width 11 pixel can eliminate most of the background noise without incurring too much losses to the main pattern.

For binarization step, the method that we choose is Otsu method. It iterates via using different threshold values, Tn, and divide the image into two set of pixels, A and B. Set A contains the pixel that valued above the threshold values, and Set B contains pixels valued below the threshold value. Both means, µA and µB, and number of pixels, WA and WB are calculated, and a final between class variance BCV is calculated as the following

BCV = WA\*WB\*(µA-µB)^2  
For n = 1,2, ...n, the Otsu’s method calculate the BCV, and at the end, it output the maximum BCV values, and which intensity level T where the maximum BCV is found. This intensity Toptimal is the global average, and by using this method, we can obtain a relatively ok binarized pictures.

## Algorithm for Locating Outline

At the boundary of the image, there is a discontinuity of pixel values. For instance, in the first binary image, the background pixels have a value of 0, and the foreground pixel is 1. At any given pixel, says, pixel a, there is only four possible scenarios with its immediate neighbor, b, which is shown below. If the sum of the immediate pairs of pixels is not zero, and the product of the immediate pairs of pixels is zero, case (2) and (3) can be identified. One more check on which pixel value is 0 would locate the blue pixel, and that pixel is marked one in the output matrix, outline(N). In this way, the outline can fully cover the contour of the foreground.

| 0 | 0 |  | 0 | 1 |  | 1 | 0 |  | 1 | 1 |  | \*a | b |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (1) | |  | (2) | |  | (3) | |  | (4) | |  |  |  |

Fig 3. Four scenarios of neighboring pairs

Based on 4 neighborhood connectivity, each pixel is to be checked twice with its immediate right neighbor, and its immediate bottom neighbor. The sweep excludes the Nth column and Nth row to avoid exceeding the bounds of the original image matrix.

| \*a | b |
| --- | --- |
| c |  |

Fig 4. Each pixel a is checked twice, a and c is checked once, followed by a and b.

## Algorithm for Segmentation

Segmentation of the characters is done by connected component algorithms. Here we use 4 connected neighborhood. The code sweeps the entire matrix space twice. The first sweep create a new matrix space segNum, which contains the labelling, followed by the second sweep that replace any connected components with different labellings.

1. First traversal: primitively label non-zero pixels

The information of connected component (CC) labelling will be stored in the array seg, which has the same dimension with img. The value of seg(i, j) indicates which connected component the pixel img(i, j) belongs to. Therefore, the range of seg(i, j) is from 1 to the total number of CCs. The zero pixels have their seg equal to 0.

During the first sweep, the order of traversing is from left to right and from top to bottom (of the image). For each non-zero pixel img(i, j), the programme checks whether the pixel at its left or right above it belongs to any existing CC. If one of them does so, seg(i, j) will follow that pixel; or if both of them belong to different CC, seg(i, j)will follow the pixel at its left. If both neighbours are zero pixels, that pixel will be labelled as part of a new CC, like the pixel in grey in the figure below, the count of the number of CCs will plus one.

| 2 |  |  |  |  | 3 |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 3 | 3 | 3 |  |
| 2 | 2 |  | 3 | 3 |  |  |
| 2 | 2 | 2 | **2** |  |  | 4 |

Fig 5. Illustration of how pixel takes value when both neighbor are non-zeros

However, when the programme reaches a pixel whose two neighbours belong to different existing CC, like the pixel in magenta in the figure above, since the traversal cannot go back, seg of the pixels in orange (labelled 3) cannot be updated from 3 to 2. That is why we need to traverse the matrix again to have all connected pixels labelled the same number.

1. Second traversal: depth-first search (DFS) to update labelling

We used depth-first search (DFS) in the second traversal, and for simplicity, recursion is used to implement the DFS. An boolean array visited is used to record whether the pixel img(i, j) has been visited. The second traversal will still check each pixel from left to right and top to bottom. When the programme reaches a pixel, which (i) belongs to a CC (seg(i, j) > 0) and (ii) has not been visited (visited(i, j) = false), the recursive function segMerge will be called, inside which those two conditions are also checked for the four neighbours of the current pixel, in the sequence of up → down → left → right. If any one neighbour at front in this sequence are found meet both conditions (i) and (ii), segMerge will be called for that neighbour, and the rest neighbours will be checked only after that segMerge returns. segMerge will return if all four neighbours of the current pixel fail to meet both conditions (i) and (ii). Therefore, in other words, the programme will go as far as possible each time exploring the connected pixels, as shown in the figure below: the yellow arrows indicate the sequence of searching starting from the pixel in magenta.

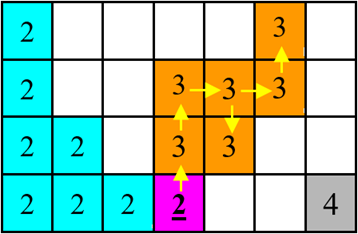


Fig 6. Illustration of how depth field search transverse in a non-visited branch

1. Count area for each CC and modify the labelling

After all connected pixels have been labelled the same number, another problem may rise: after all pixels labelled “3” have been updated to “2”, as shown below, the pixels originally labelled “4” are not updated to “3”, which results in a gap between 2 and 4. On the other hand, however, segment 3 still exists, but it is just empty and has an area of 0.

| 2 |  |  |  |  | 2 |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 2 | 2 | 2 |  |
| 2 | 2 |  | 2 | 2 |  |  |
| 2 | 2 | 2 | 2 |  |  | 4 |

Fig 7.Illustration of segment when function updateSegNum is done

Therefore, in the next step, the programme will traverse the entire matrix again to count the area of each segment, which is just the total number of pixels with that label. After the pixel count of all segments has been completed, the programme will discard those segments with the area less than a threshold. For image 1, the threshold is 0, because there is no noise, and there are only 6 CCs. For image 2, however, since some noise cannot be fully eliminated through opening, like the white block in the figure below, which is image 2 after opening and thresholding, we need a higher threshold. There are a large number of choices for the threshold value, because it will work as long as the area of all non-noise segments are larger than it and all noise segments have an area smaller than it. Thus, we chose 100 as the area threshold value.



The programme establishes a mapping between the old and new segmentation labelling with an array updateSegNum which has the dimension of 1 × number of segments in the first traversal. The index n of the array is the label after the second traversal (all connected pixels have the same label, but labelling is not continuous), and the value of updateSegNum(n) is the final label. Hence, in the fourth traversal, if seg(i, j) > 0, seg(i, j) will be updated to updateSegNum(seg(i, j)). By now, all the characters have been labelled 1, 2, 3, …, nSeg, and all the noise has been eliminated.

1. Determine the bounding rectangles

The bounding rectangle of a character is a rectangle that all pixels belong to that character are always bounded by this rectangle. In other words, once we know the minX, minY, maxX, and maxY among all pixels of a character, the bounding rectangle can be fully defined. Therefore, the programme traverses seg again, and records the minX, minY, maxX, and maxY for each segment.

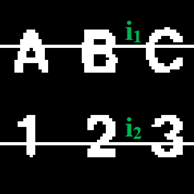
To store the bounding rectangle information, a self-defined class, RectBoundary, is used, whose members are nothing but four integers minX, minY, maxX, and maxY. The array rects\_t, which is an array of RectBoundary type variables, is used to store the coordinates of bounding rectangle vertices for each segment. One problem here is that in image 2, there are three connected character pairs “D4”, “80”, and “00”. Since character recognition algorithms using machine learning are not expected to be involved in this project, we manually divided the bounding rectangle for those pairs into two rectangles.

1. Rearrange the order of the characters

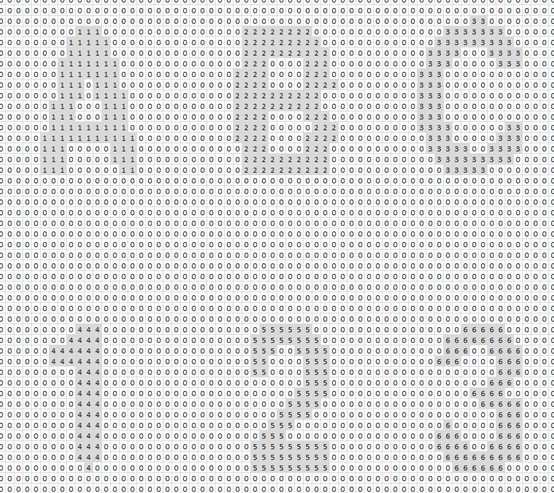
By now, we have determined the bounding rectangle for each segment, but the order is random. I.e. using image 1 as an example, rectangle 1, 2, …, 6 may not corresponds to “A”, “B”, “C”, “1”, “2”, and “3”. Therefore, the next task is to have all the characters be labelled 1, 2, …, nSeg, from left to right and up to down.

The first step is to check the elements of two rows of seg: i1 = N/4 and i2 = 3\*N/4 for image 1 and i1 = H/3 and i2 = 2\*H/3 for image 2, as shown below. The rows to be checked have been highlighted by the white lines. An array reEnum will be used to store the mapping between the old and the new order, whose function is similar to the aforementioned array updateSegNum; and a counter is used to record the number of the segments that the programme has detected.

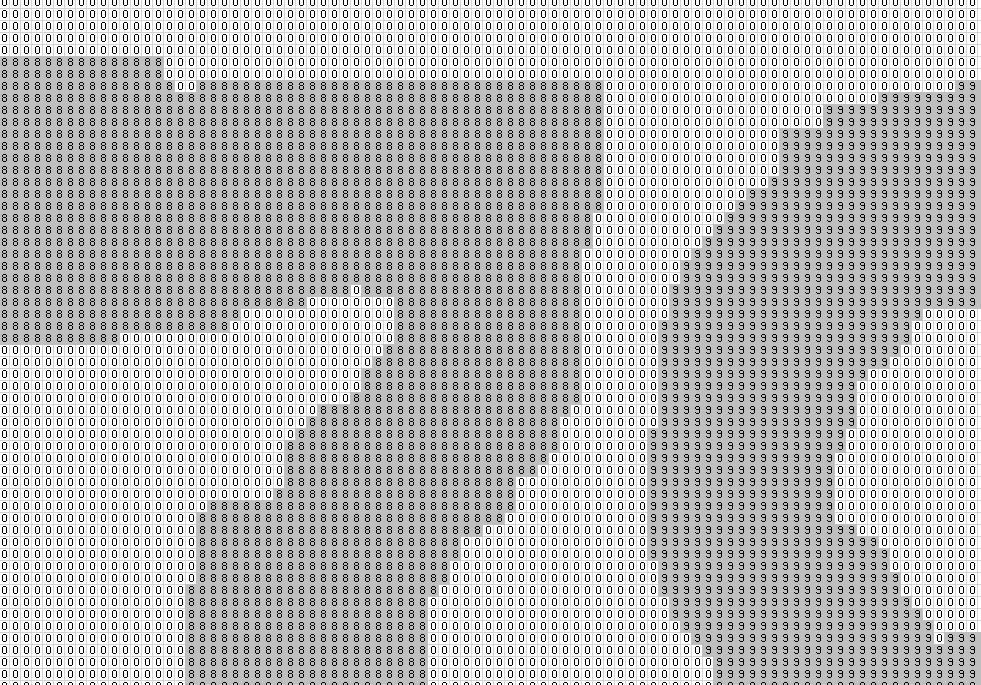
For example, for image 1, let j denote the horizontal coordinate, which goes from 1 to 64. When a pixel with seg(i, j)> 0 is encountered, counter = counter + 1, reEnum(seg(i, j)) = counter, and x = rects\_t(i, j)).maxX + 1. The last statement helps j directly jump into the bounding rectangle of the next segment, so that a plenty of further check can be saved.

The programme then traverses seg another time and update the value of seg for all non-zero pixels with seg(i, j) = reEnum(seg(i, j)). The figure below shows the value of the elements of seg for image 1 after all the steps above. It can be seen that all the segments have been properly labelled in the correct sequence.



Everything is the same for image 2, but since the size of image 2 is much larger than image 1, only part of seg of image 2 are shown for illustration.



By now, the programme has known the positions of each character, based on which, we can easily display the characters in any order we want, and the rotation operation

## Algorithm for Segment Rotation

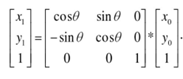
According to mathematics theorem, while rotating a point A(x­0, y0) in a Cartesian coordinate system with a certain degree θ through point O (0,0), the new coordinate, A1 will become (X1,Y1), with

X1=x0 cosθ+y0 sinθ.

Y1=-x0 sinθ+y0 cosθ.

In matrix format, the equation will be

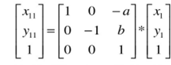
Therefore, the rotated matrix’s size will also become



W2=w1 cosθ+h1 sinθ.

H2=w1 sinθ+h1 cosθ.

If the matrix is moved to a cartesian coordinate with origin (a, b) instead of O (0,0), the relation between new coordinate and the old one will become



while (x11, y11) are the new coordinate of point A and (x1, y1) are the old coordinate.

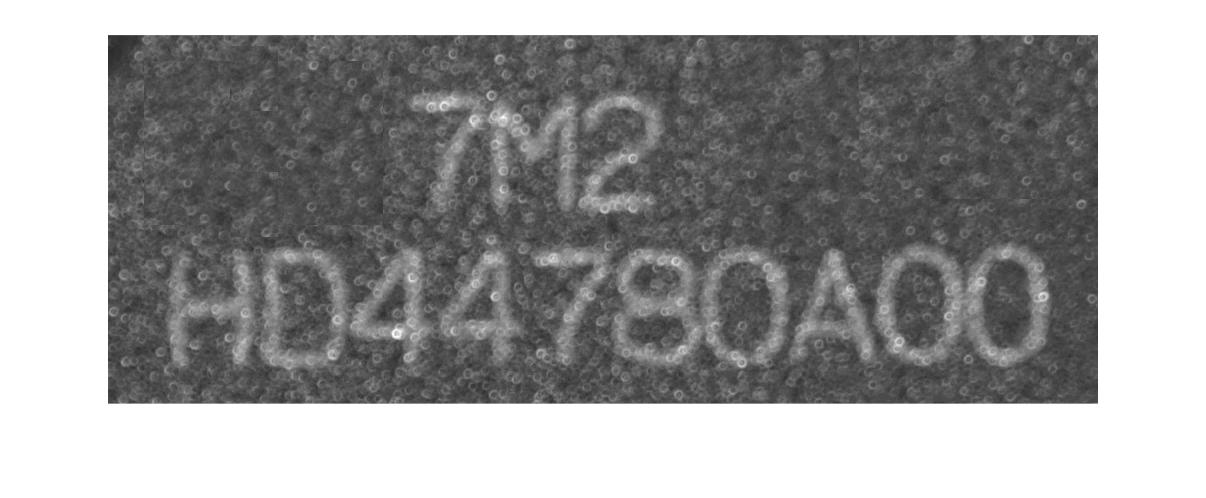
Then these equations are combined, A0 (x0, y0) of matrix rotating through a point A (a, b), where A is the centroid of of the rotated picture with a relation of



The original coordinates of each points are all integer, but after multiplication, the new coordinates may not be integers. Therefore, there will be hollow points in the rotated picture. To solve this problem, a nearest interpolation algorithm is needed. In order to calculation the grey level of a point P (x0, y0), calculate the nearest four points with integer coordinates of P, then assign the grey level of nearest point to P.

## Algorithm for Thinning Operation

The thinning algorithm used can be stated as:

1. Find a pixel location (i, j) where the pixels in the image match those in template T1, [0,1,1]’. With this template all pixels along the top of the image are removed moving from left to right and from top to bottom.

2. If the central pixel is not an endpoint, and has connectivity number = 1, then mark this pixel for deletion.

Endpoint pixel: A pixel is considered an endpoint if it is connected to just one other pixel. That is, if a black pixel has only one black neighbour out of the eight possible neighbours.

3. Repeat steps 1 and 2 for all pixel locations matching T1.

4. Repeat steps 1-3 for the rest of the templates: T2, which is T1’, T3, which is [1,1,0]’, and T4=T3’.

T2 will match pixels on the left side of the object, moving from bottom to top and from left to right. T3 will select pixels along the bottom of the image and move from right to left and from bottom to top. T4 locates pixels on the right side of the object, moving from top to bottom and right to left.

5. Set to white the pixels marked for deletion.

# **III. Results**

## Screenshots for Image 1

| Task No. | Task Descriptions | Resultant Screenshots | Comments |
| --- | --- | --- | --- |
| 1 & 2 | Display & binarize the image |  |  |
| 3 | Outline |  |  |
| 4 | Segmentation |  | Each segmented part is randomly colored. |
| 5 | Rotation by 90 degree clockwise |  |  |
| 6 | Rotation by 30 degree counterclockwise |  |  |
| 7 | Thinning of image |  |  |
| 8 | Realignment |  |  |

## 

## Screenshot for Image 2

| Task No | Task | Screen Shot | Comments |
| --- | --- | --- | --- |
| 1. | Display Image |  |  |
| 2 | Binarize Image |  |  |
| 3 | Outline |  |  |
| 4 | Segmentation |  | Each segmentation body is colored with a different color |
| 5 | Rotation by 90 degree clockwise |  |  |
| 6 | Rotation by 30 degree counterclockwise |  |  |
| 7 | Thinning |  |  |
| 9 | Realignment |  |  |

# **IV. Assessment of Methods**

## Thresholding

Thresholding is essentially re-labelling the pixel values with reference to a predetermined threshold value. Converting colored image to grayscale is a multi-band thresholding, and binarization is a single thresholding problem. In this project, we focus on the basic binarization process for image 2.

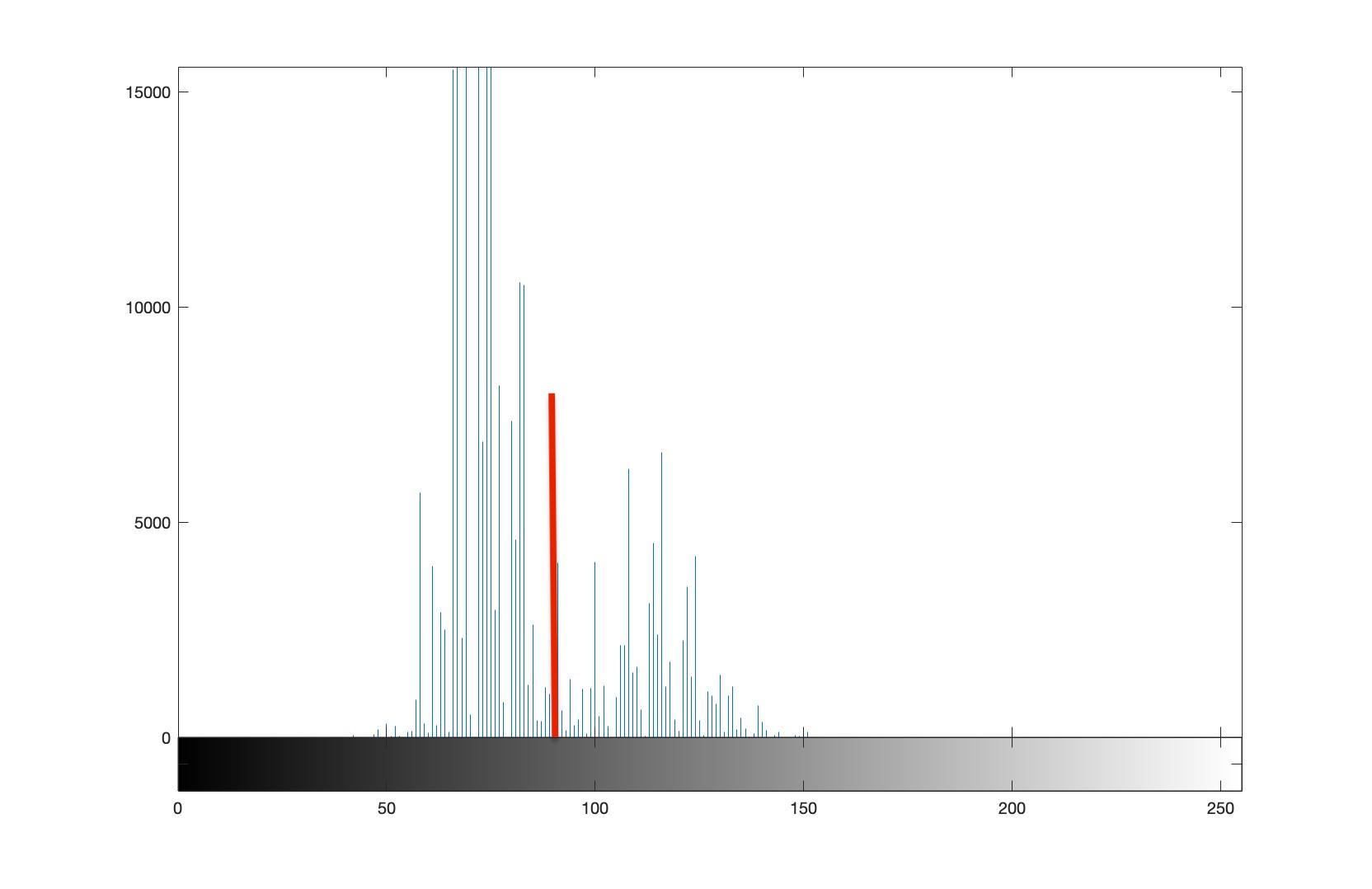


Fig. a histogram of Image 2, and the red line indicate a rough estimate of global average

Image 2 has a roughly bimodal distribution of pixel, where the dark background concentrates around pixel value 75, and the white body of alphabets cluster around 125. For simpler case like this, it is favorable to calculate a global average of the pixel values, where the red line stands, and threshold the image with that global average.

We have compared the results obtained by the inbuilt imbinarize and our own binarization function, The results are highly alike. Referring back to the histogram of pixel values, the distribution of pixels allow global thresholding method work nicely. If the distribution of pixels is more uniform, more refined methods taking into account of the pixel values in the locality must be considered. Being an inbuilt function, imbinarize has options ‘adaptive’ to accommodate such conditions, while our own written thresT is equipped with such options.



Fig Image obtained by inbuilt imbinarize after opening



Fig. Binary image from Otsu’s method after opening

## Segmentation

Segmentation in the project involves two parts. The first is to perform a connected component algorithm, while the second is to determine the rectangular boxes bounding each character pattern. The discussion here concerns the first part only, i.e. algorithm for labelling connected components.

The DFS-based algorithm we used has identical output with the Space-Efficient Two-Pass Algorithm using Local Equivalence Table we have learnt in class, and their time complexity are also almost the same (both are O(W\*H)). In terms of simplicity, conciseness, and space complexity, nonetheless, the algorithm we have learnt in class is better than our own algorithm: the use of local equivalence table makes that two-pass algorithm in-place, while ours needs auxiliary matrices with the same size as the original image.

However, one reason we developed our own method is that we found it practically not easy to implement the local equivalence table with MATLAB. Unlike C++, MATLAB does not provide something like the Standard Template Library (STL), which provides templates for a wide range of data structures like vector and set, with which it would be easy to implement the idea of the equivalence table: we just need a vector of sets, say, vector<set<int>> equiTable. Then each equiTable[i] represents the equivalence table for the i-th set of equivalent pairs, and we can use equiTable[i].insert(n) to insert the new equivalent pair into the i-th table. In contrast, if we want to implement the equivalence table in MATLAB, we can only use arrays, which can be quite inefficient. Therefore, we developed our own DFS-based algorithm.

## Thinning

Another method to use is a method called Zhang-Suen thinning algorithm (T.Y Zhang and C. Y. Suen, 1984):

Algorithm

Assume black pixels are one and white pixels zero, and that the input image is a rectangular N by M array of ones and zeroes.

The algorithm operates on all black pixels P1 that can have eight neighbours.

The neighbours are, in order, arranged as:

| P9 | P2 | P3 |
| --- | --- | --- |
| P8 | P1 | P4 |
| P7 | P6 | P5 |

Obviously the boundary pixels of the image cannot have the full eight neighbours.

Define A(P1) = the number of transitions from white to black, (0 -> 1) in the sequence P2, P3, P4, P5, P6, P7, P8, P9, P2.

Define B(P1) = The number of black pixel neighbours of P1. (= sum(P2 .. P9) )

All pixels are tested and pixels satisfying all the following conditions (simultaneously) are just noted at this stage.

(0) The pixel is black and has eight neighbours

(1) 2<=B(P1) <=6

(2) A(P1) = 1

(3) At least one of P2 and P4 and P6 is white

(4) At least one of P4 and P6 and P8 is white

After iterating over the image and collecting all the pixels satisfying all step 1 conditions, all these condition satisfying pixels are set to white.

2. All pixels are again tested and pixels satisfying all the following conditions are just noted at this stage.

(0) The pixel is black and has eight neighbours

(1) 2<=B(P1) <=6

(2) A(P1) = 1

(3) At least one of P2 and P4 and P8 is white

(4) At least one of P2 and P6 and P8 is white

After iterating over the image and collecting all the pixels satisfying all step 2 conditions, all these condition satisfying pixels are again set to white.

3. Iteration

If any pixels were set in this round of either step 1 or step 2 then all steps are repeated until no image pixels are so changed.

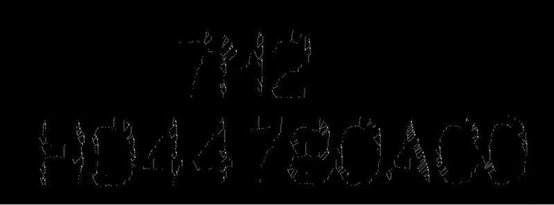


Fig. Resultant image from Zhang-Suen method.

The result displays as below image which is shown as above image, which is relatively poor compared to the Stentiford method. Therefore we choose the first algorithm called stentiford thinning algorithm.

# **V. Conclusion**

Similarity and Differences:

To achieve the image processing as required, both images should be transferred to grey level image first. The first difference in processing with the two image is the image opening step. For the first image, the txt file is converted to an image format by changing all the point value greater than 1 in the in the txt file to bright pixels while remain the 0 values to be dark pixels. The second image, bmp file is an image format, but the image is still needed to be converted to a binary image. Therefore, the similarity is to convert to grey level and binary image for these two images. The difference is to convert the first txt file to an image.

The algorithm for processing the other steps are similar while the second image needs to be refined as there are obvious large noise points in the image. The traditional algorithm such as using filters does not provide a relatively good processing result as the noise points are too large. Therefore, a method by using the opening morphology is used as a function imopen. The structuring element is chosen to be a 11 pixels square to have a relatively good removal of these noise points with input: SE=strel(‘square’,11);The number 11 is chosen after roughly measuring these noise points’ dimension and eliminate these noises.

# **V. References**

T.Y.Zhang and C.Y.Suen (March 1984), ‘*A Fast Parallel Algorithm for Thinning Digital Patterns*’, Image Processing and Computer Vision.